



快速解答常见问题

OpenCV 2 计算机视觉 编程手册

(加) Robert Laganière 著
张 静 译

cvchina.info 推荐
精通实时计算机视觉编程的 50 多个秘诀



科学出版社

OpenCV 2 计算机视觉编程手册

这是一本循序渐进的计算机视觉指导手册，基于 OpenCV 2 代码库中包含高级特性的 C++ 接口。

本书介绍了 OpenCV 2 中众多的视觉算法。你将学会如何读、写、创建及操作图像，领略图像分析中常用的技术，并了解如何使用 C++ 高效实现。

本书内容

- 遍历图像处理每一个像素
- 读写视频序列
- 运用成熟的面向对象技巧创建高级的视觉应用
- 使用直方图增强一幅图像，或增强图像的某些部分
- 使用数学形态学及图像滤波进行图像变换及图像分割
- 利用图形几何学匹配同一个符合场景中的不同视角
- 处理视频帧，从而跟踪特征点并识别动态物体
- 检测图像中的点、线条、轮廓及物体

本书程序代码及彩图下载：

<http://www.sciencep.com/downloads/>

科学出版社 东方科龙公司

联系电话：010-82840399

E-mail：boktp@mail.sciencep.com

有关网址：<http://www.okbook.com.cn>

销售分类建议：计算机/程序设计/图像处理

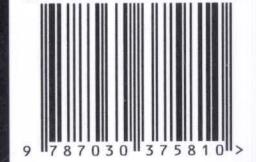


本书特色

- 简明易懂的叙述格式
- 一系列最重要的任务和问题
- 精心编制的针对高效解决问题的指导
- 对代码的清晰解释
- 举一反三的应用

www.sciencep.com

ISBN 978-7-03-037581-0



9 787030 375810 >

定 价：45.00 元

OpenCV 2

计算机视觉编程手册

[加] Robert Laganière 著

张 静 译

科学出版社

图字：01-2012-0919号

内 容 简 介

本书以案例的形式介绍OpenCV 2.X的新特性和C++新接口，案例中包含具体的代码与详细的说明。本书很好地平衡了基础知识与进阶内容，要求读者具有基础的C++知识。

本书既适合想要学习计算机视觉的C++初学者，也适合专业的软件开发人员。本书可作为高等院校计算机视觉课程的辅助教材，也可以作为图像处理和计算机视觉领域研究人员的参考手册。

图书在版编目（CIP）数据

OpenCV 2计算机视觉编程手册/ (加) Robert Laganière著; 张静译.—北京: 科学出版社, 2013.7

书名原文: OpenCV 2 Computer Vision Application Programming Cookbook
ISBN 978-7-03-037581-0

I.O… II.①R… ②张… III.图像处理软件-程序设计-技术手册
IV.TP391.41-62

中国版本图书馆CIP数据核字 (2013) 第111236号

责任编辑: 喻永光 杨 凯 / 责任制作: 魏 谦

责任印制: 赵德静 / 封面制作: 段淮沱

北京东方科龙图文有限公司 制作

<http://www.okbook.com.cn>

科学出版社出版

北京东黄城根北街16号

邮政编码: 100717

<http://www.sciencep.com>

北京源海印刷有限责任公司 印刷

科学出版社发行 各地新华书店经销

*

2013年7月第一 版 开本: 787×960 1/16

2013年7月第一次印刷 印张: 16 1/4

印数: 1—3 500 字数: 289 000

定价: 45.00元

(如有印装质量问题, 我社负责调换)

译者序

我和OpenCV的故事

一切的起源是OpenCV_1.0.exe的安装包，那时我电脑中安装的还是VC++6.0绿色版，我以为计算机视觉只是图像处理的别名。当我运行samples/facedetect时惊呆了，笔记本自带的摄像头居然实时地识别出我的脸。后来我意识到这句话说错了，其实是检测到了我的脸。人脸识别指的是计算机知道我是谁，这个功能在当时的OpenCV是不支持的。另外一个让我震惊的是samples/inpaint.exe，字面意思是图像修补，即智能地填补缺失的像素信息，那会儿我想这可以用来去马赛克。后来想明白了，信息是不会凭空出现的，如果你事先不知道马赛克背后可能是什么，那么无法去恢复它。

2008年上海双年展上，我看到一款摄像头互动的艺术作品《上海，我能邀你跳支舞吗？》，参展者在摄像头前通过身体来影响投影屏幕中的虚拟建筑。这是我从没想象过的人机交互方式，我意识到程序的操控设备可以脱离鼠标键盘，进入到现实的空间中。回来后我用OpenCV自制了一个用双手控制的虚拟打鼓游戏，录好视频后放在优酷上，启发了一些人。这便是最原始的体感，在硬件上只依赖一个摄像头。一年后的E3展上，Kinect以Project Natal的名称为世人所知。

提到摄像头互动，自然不能不说基于CCV的多点触摸，国内外不少公司都在凭借这款开源的软件接活。而CCV最核心的摄像头跟踪模块完全依赖OpenCV，所做的无非是阈值化加上寻找连通区域。因为打开了OpenCV这扇门，我接触到截然不同的编程世界。研究CCV的间隙我还学习了openFrameworks和Processing，今年我还有一本Processing的翻译书籍要出版，这是另外一个故事。

之后从事了游戏行业，基本没机会用OpenCV，直到一个游戏项目中的字体贴图丢失了坐标信息，需要通过肉眼手动定位。这很不科学，于是我拾起OpenCV，阈值化加上寻找连通区域，输入一副图像，输出字体的包围盒坐标，同事纷纷觉得不可思议。计算机视觉的魅力便在此，从看似无关的像素中能够提取信息。这不是唯一一次在工作中使用OpenCV，之后我得到了一份增强现实、图像检索等领域的开发工作，可以全

职地进行OpenCV的开发。共事的有国内著名CV网站cvchina.info的站长张小军，本书的第2章也是由他翻译的，对于他在工作中的指点在此一并感谢。

OpenCV的用途很广，除了上述提到的，还有视频监控、双目视觉、机器学习、Kinect深度信息处理、照片美化、图像拼接、GPU加速等功能。但是OpenCV的书籍并不多，在本书面世前市场上只有一本讲述1.0接口的*Learning OpenCV*。本书的意义在于，讲述了2.0后出现的C++接口以及大量的新功能。如果你看过*Learning OpenCV*，想学习新接口使用，那么推荐你阅读这本书。如果你没有接触过OpenCV，想直接跳过C接口的函数，那么更加推荐你阅读本书。

最后，感谢陪伴我鼓励我的妻子M，如果没有你，我和OpenCV的故事到第二段就结束了，译者序也必定由他人撰写。

张 静

2013年4月25日 上海

前 言

图像与视频在当今的数字世界可谓无处不在，随着运算能力强劲而又实惠的计算设备的问世，创建复杂的图像应用从未像今天这般容易。市面上有众多的软件和库可用于操作图像与视频，但是对于期望自己开发软件的人而言，OpenCV库是一款必备的工具。

OpenCV（Open Source Computer Vision）是一个开放源代码的图像及视频分析库，它包含500多个优化过的算法。自1999年问世以来，它已经被计算机视觉领域的学者和开发人员视为首选工具。OpenCV最初是由Intel的一个小组进行开发的，领头人是Gary Bradski，作为视觉领域研究的先锋，他推动了许多CPU密集的视觉应用。在发布一系列Beta版本后，1.0版本终于在2006年面世。第二次重要的版本发布是2009年的OpenCV 2，它带来了重要的变化，尤其是包括本书中使用的崭新的C++接口。写作本书时，最新的版本是2.2（2010年10月）。

本书覆盖了库中的许多特性，并且展示了如何使用它们完成特定的任务。我们的目标并非事无巨细地讲述OpenCV提供的函数与类，而是教你零基础进行视觉开发的必备知识。本书中还解释了图像分析中的基本概念，并且对计算机视觉中的一些重要算法进行了介绍。

本书对读者而言是接触图像处理与视频分析的一次良机，但这也只是开端。OpenCV一直处于改进与扩展功能的阶段，查询在线文档可以了解最新的进展：<http://opencv.org/>。

本书内容

第1章 接触图像：介绍OpenCV库，并展示如何使用微软Visual C++与Qt开发环境运行简单的应用。

第2章 操作像素：解释如何读取图像，将使用不同的方法扫描图像以进行逐像素操作。你也将学习到如何定义图像中的兴趣区域。

第3章 基于类的图像处理：展示多种面向对象的设计模式，将帮助你更好地构建

计算机视觉应用。

第4章 使用直方图统计像素：介绍如何计算图像的直方图，以及利用直方图进行图像修改。利用直方图可以实现图像分割、物体检测以及图像检索。

第5章 基于形态学运算的图像变换：探索形态学变换的概念，展示了不同的形态学操作子以及它们在检测边缘、角点与线段中的用途。

第6章 图像滤波：介绍频域分析及图像滤波的原理。展示了低通、高通滤波器在图像中的应用，并提出两个图像微分算子——梯度和拉普拉斯算子。

第7章 提取直线、轮廓及连通区域：专注于讲解对几何图像特征的检测，解释如何从图像中提取轮廓、线条以及连通区域。

第8章 检测并匹配兴趣点：介绍不同的兴趣点探测算法，还解释如何计算特征点的描述符，以及如何使用描述符来匹配图像。

第9章 估算图像间的投影关系：分析图像形成中涉及的不同关系，还探讨了同一个场景的两幅图像之间存在的投影关系。

第10章 处理视频序列：提供一个框架来读写视频并且处理每一帧，还展示了如何在相邻帧之间跟踪兴趣点，以及如何提取相机前移动的前景物体。

本书读者

如果你是一个C++程序员新手，并且想学习如何使用OpenCV库来构建计算机视觉应用，那么本书非常适合你。同时，它也适合希望学习计算机视觉编程的专业开发人员，可以作为大学计算机视觉课程的学生用书。对于图像处理和计算机视觉领域的研究生及研究人员而言，它也是一本极佳的参考手册。本书很好地综合了基础知识与进阶内容。当然，这要求读者掌握一定的C++技能。

本书约定

在这本书中，你会发现许多样式的文字，它们有着不同的含义。下面是一些样式的范例，以及它们的用途。

代码中的单词如下显示：“通过使用`include`指令我们能够引入其他的上下文。”

代码块的样式则是：

```
//获取迭代器  
cv::Mat<cv::Vec3b>::const_iterator it=  
image.begin<cv::Vec3b>();
```

当我们希望将读者的注意力集中到某个代码块的部分时，相关的代码行或者文字将设为粗体：

```
//转换为Lab色彩空间  
cv::cvtColor(image, converted, CV_BGR2Lab);  
  
//获取转换后图像的迭代器  
cv::Mat<cv::Vec3b>::iterator it=
```



提示和技巧以这样的样式显示。

读者反馈

我们始终欢迎读者的反馈意见。我们渴望知道你对于本书的看法，喜欢哪些内容或者不喜欢哪些内容。你的真实感受，对于我们开发读者切实需要的图书十分重要。

如果你有反馈意见，请发送电子邮件至 feedback@packtpub.com，并在邮件主题中注明你评论的书名。

如果你希望我们出版某方面的书，请在 www.packtpub.com 填写“SUGGEST A TITLE”表格，或发送邮件至 suggest@packtpub.com。

如果你对某个主题有经验或者有兴趣，愿意撰写或参与撰写一本书，请查看 www.packtpub.com/authors 页面中的“作者指南”。

客户支持

现在，你已经拥有一本Packt出版的书了，为了让你的付出得到最大的回报，请注意以下事项。

下载资源包

你可以从你在 <http://www.packtpub.com> 的账户中，下载已购买的所有 Packt 书籍的相关文件。如果是在别处购买的这本书，你可以访问 <http://www.packtpub.com/support>，然后填上邮箱，让服务器直接把这些文件通过电子邮件发给你。

勘 误

虽然我们会尽全力确保本书内容的准确性，但错误仍时有发生。如果你发现了本书中的错误（包括文字和代码错误），而且愿意告诉我们，我们将十分感激。这样做不仅可以减少其他读者的挫败感，也有助于本书再版时的改进。如果你发现任何错误，都可以访问 <http://www.packtpub.com/support>，选择相应的书，以勘误表的形式报告具体错误。一旦得到验证，你的意见将被采纳，勘误表也会被上传到我们的网站，或者添加到已有的勘误表中。已有的勘误表可以在 <http://www.packtpub.com/support> 查阅。

举报盗版

互联网上涉及各种媒体的盗版行为始终存在。Packt 非常重视版权的保护和许可。无论您在互联网上发现对我们的作品的任何形式的盗版，请告诉我们其链接或网址，以便我们采取相应的补救措施。

请将涉嫌包含盗版资料的链接发送至 copyright@packtpub.com。

感谢你保护作者的权益及我们继续为你提供有价值的内容的能力。

疑问解答

有关于本书的任何问题，你都可以通过邮箱 questions@packtpub.com 联系我们，我们将尽最大的努力解决。

目 录

第 1 章 接触图像

1.1 引言	1
1.2 安装OpenCV库	1
1.3 使用MS Visual C++创建OpenCV工程	4
1.4 使用Qt创建OpenCV项目	12
1.5 载入、显示及保存图像	18
1.6 使用Qt创建GUI应用	22

第 2 章 操作像素

2.1 引言	29
2.2 存取像素值	30
2.3 使用指针遍历图像	33
2.4 使用迭代器遍历图像	40
2.5 编写高效的图像遍历循环	43
2.6 遍历图像和邻域操作	46
2.7 进行简单的图像算术	50
2.8 定义感兴趣区域	53

第 3 章 基于类的图像处理

3.1 引言	57
3.2 在算法设计中使用策略 (Strategy) 模式	57
3.3 使用控制器 (Controller) 实现模块间通信	63
3.4 使用单件 (Singleton) 设计模式	67
3.5 使用模型—视图—控制器 (Model—View—Controller) 架构设计应用程序 ..	70

3.6 颜色空间转换	73
------------------	----

第 4 章 使用直方图统计像素

4.1 引言	77
4.2 计算图像的直方图	77
4.3 使用查找表修改图像外观	84
4.4 直方图均衡化	88
4.5 反投影直方图以检测特定的图像内容	89
4.6 使用均值漂移 (Mean Shift) 算法查找物体	95
4.7 通过比较直方图检索相似图片	99

第 5 章 基于形态学运算的图像变换

5.1 引言	103
5.2 使用形态学滤波对图像进行腐蚀、膨胀运算	103
5.3 使用形态学滤波对图像进行开闭运算	107
5.4 使用形态学滤波对图像进行边缘及角点检测	110
5.5 使用分水岭算法对图像进行分割	116
5.6 使用GrabCut算法提取前景物体	121

第 6 章 图像滤波

6.1 引言	125
6.2 使用低通滤波器	126
6.3 使用中值滤波器	130
6.4 使用方向滤波器检测边缘	132
6.5 计算图像的拉普拉斯变换	138

第 7 章 提取直线、轮廓及连通区域

7.1 引言	143
7.2 使用Canny算子检测轮廓	143

7.3 使用霍夫变换检测直线	146
7.4 用直线拟合一组点	156
7.5 提取连通区域的轮廓	160
7.6 计算连通区域的形状描述符	164

第 8 章 检测并匹配兴趣点

8.1 引言	167
8.2 检测Harris角点	167
8.3 检测FAST特征	177
8.4 检测尺度不变的SURF特征	180
8.5 描述SURF特征	185

第 9 章 估算图像间的投影关系

9.1 引言	189
9.2 相机标定	191
9.3 计算一对图像的基础矩阵	198
9.4 使用随机采样一致算法 (RANSAC) 进行图像匹配	202
9.5 计算两幅图之间的单应矩阵	211

第 10 章 处理视频序列

10.1 引言	215
10.2 读取视频序列	215
10.3 处理视频帧	219
10.4 写入视频序列	228
10.5 跟踪视频中的特征点	233
10.6 提取视频中的前景物体	239

第 1 章

接触图像

本章主要探讨以下内容：

- ◆ 安装 OpenCV 库；
- ◆ 使用 MS Visual C++ 创建 OpenCV 工程；
- ◆ 使用 Qt 创建 OpenCV 项目；
- ◆ 载入、显示及保存图像；
- ◆ 使用 Qt 创建 GUI 应用。

1.1 引言

本章将讲授 OpenCV 的基础元素，并将带领你完成最基本的操作：读取、显示及保存图像。在着手开发前，需要先安装库。本章的第一则秘诀将解释这个简单的过程。

你还需要一个顺手的开发环境（IDE），我们推荐微软的 Visual Studio 及开源的 C++ 开发工具 Qt。两则独立的秘诀将分别介绍如何使用这两个工具建立项目，你也可以使用其他 C++IDE。事实上，在本书中展示的项目都不依赖特定的环境或操作系统，你可以自由选择。但是，需要确保编译后的 OpenCV 库与你使用的编译器与操作系统相兼容。如果你遇到怪异的结果，或是应用程序莫名地崩溃，那么很有可能是因为不兼容。

1.2 安装OpenCV库

OpenCV 是开源的计算机视觉代码库，它既可以用于学术上也可以用于商业应用。只要遵守 BSD 协议，便可自由地使用、发布及修改。本则秘诀将介绍如何进行库的安装。

准备工作

OpenCV 的官方网址是 <http://opencv.org/>，在这里你可以找到最新发布的版本、在线文档以及大量有价值的资源。

实现方法

从网站中可以找到对应平台（Linux/Unix/Mac 或 Windows）的下载页，下载后需要解压到一个文件夹中（如 OpenCV2.2）。解压完成后，你将看到一组文件夹，其中 doc 文件夹中包含的是文档，而 include 文件夹中是所有的头文件，modules 文件夹中包含所有的源文件（别忘了它是开源的），而 samples 文件夹中则是许多简短的学习用范例。

如果在 Windows 系统中使用 Visual Studio 进行开发，你可以选择下载可执行的安装程序。运行安装程序不仅会复制源代码库，还会安装所有预编译的二进制文件。在这种情况下，你可以直接开始使用 OpenCV，否则还需要一些额外的步骤。

为了在你选择的环境中进行 OpenCV 开发，需要使用合适的 C++ 编译器生成二进制的库文件。你需要使用 CMake 工具，它位于 <http://www.cmake.org/>，这是另外一个开源的软件工具，基于平台无关的配置文件来控制编译的过程。你可以使用命令行运行它，但是使用图形化界面更方便些。使用后者，你只需要指定包含 OpenCV 库的文件夹及包含二进制文件的文件夹，然后点击 Configure 按钮以选择编译器的类型（这里选择的是 Visual Studio 2010），并再一次点击 Configure，如图 1.1 所示。

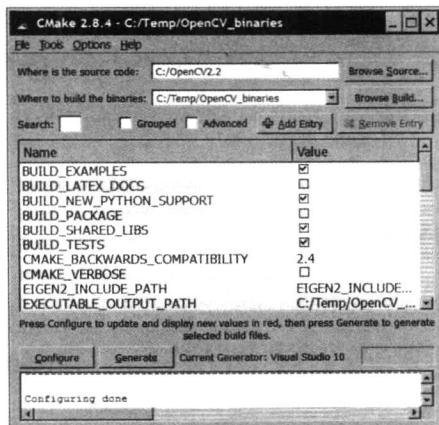


图 1.1

通过点击 Generate 按钮可以生成 makefiles 或是 workspace 文件。这些文件将允许你进行库的编译过程，这是安装的最后一步。

编译好库后，便可以进行开发了。如果你使用的是 Visual Studio 这种 IDE，那么你只需要打开顶层的 Solution 文件，这是 CMake 为你创建的。然后，开始 Build Solution 命令。在 Unix 环境中，你将使用 makeutility 命令来操作 makefiles 文件。

如果一切正常，那么你指定的文件夹中应该已经包含编译后的 OpenCV 库了。除了之前提到的文件夹，这个文件夹中还将包含一个 bin 文件夹。你可以把所有文件移动到合适的位置（如 c:\OpenCV2.2），并且将 bin 文件夹添加到系统路径中（在 Windows 系统中，通过打开控制面板中的系统（System）工具，在高级（Advanced）标签页中你将找到环境变量（Environment Variables）按钮）。

作用原理

自版本 2.2 开始，OpenCV 库便被划分为多个模块。这些模块编译成库文件后，位于 lib 文件夹中。

- ◆ opencv_core 模块，包含核心功能，尤其是底层数据结构和算法函数。
 - ◆ opencv_imgproc 模块，包含图像处理函数。
 - ◆ opencv_highgui 模块，包含读写图像及视频的函数，以及操作图形用户界面函数。
 - ◆ opencv_features2d 模块，包含兴趣点检测子、描述子以及兴趣点匹配框架。
 - ◆ opencv_calib3d 模块，包含相机标定、双目几何估算以及立体视觉函数。
 - ◆ opencv_video 模块，包含运动估算、特征跟踪以及前景提取函数与类。
 - ◆ opencv_objdetect 模块，包括物体检测函数，如脸部与行人检测。
- 库中还包含其他的工具模块，如机器学习（opencv_ml）、计算几何（opencv_flann），第三方代码（opencv_contrib）、废弃的代码（opencv_legacy）及 GPU 加速过的代码（opencv_gpu）。这些模块都有一个单独的头文件（位于 include 文件夹）。典型的 OpenCV C++ 代码将包含所需的模块，本书推荐的声明方式如下：

```
#include<opencv2/core/core.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/highgui/highgui.hpp>
```

如果你看到 OpenCV 代码以如下方式开始：

```
#include "cv.h"
```

这是因为它使用了旧的代码方式，那时库还没有被划分为模块。

扩展阅读

你可以得到最新的代码，它们位于 OpenCV 的 GIT 服务器中：

```
git://code.opencv.org/opencv.git
```

从中你将会找到大量的范例，可以帮助你进行学习并且告诉你一些开发技巧。

1.3 使用MS Visual C++创建OpenCV工程

使用 Visual C++ 可以轻易地为 Windows 创建 OpenCV 应用。你能创建简单的控制台应用或是拥有图形用户界面的复杂应用。此处我们选择最简单的控制台应用。我们将使用 Visual Studio 2010，但是同样的步骤可以用于其他版本的微软 IDE，因为它们有着类似的菜单和选项。

如果你是第一次运行 Visual Studio，那么可以进行设置，将 C++ 视为默认开发环境。这样，当你启动时将处于 Visual C++ 模式。

假设你将 OpenCV 放置在 C:\OpenCV2.2 文件夹中，正如之前的秘诀所介绍的。

准备工作

使用 Visual Studio 进行开发时，很重要的一点是明白解决方案（Solution）与工程（Project）之间的区别。本质上说，解决方案是由多个工程组成的（每个工程是一个独立的软件模块，如一个程序和一个代码库）。这样做好处是解决方案可以共享文件和代码库。通常，你为解决方案创建一个主文件夹，包括所有的工程文件夹。但是，你也可以将解决方案与工程放置于同一个文件夹内，这对于仅包含单个工程的解决方案来说是个惯例。当你熟悉 Visual C++ 并且创建更多复杂的应用之后，你应该使用包含多个工程的解决方案结构。

同时，当你编译和执行 Visual C++ 工程时，你可以在两种不同的配置下进行：Debug 及 Release。Debug 模式用于调试应用程序，这是个受保护的运行环境，它将告诉你程序是否含有内存泄露，在运行时也能对特定函数的结果进行检查。然而，它

生成的可执行文件运行较慢。因此，当你的应用经过测试准备投入使用时，你将在 Release 模式下进行编译，这将生成供最终用户使用的可执行文件。经常会发生 Debug 模式下运行正常而 Release 模式出错的情况，这时你需要进行更多测试以找到潜在的错误源。Debug 及 Release 模式并不仅在 Visual C++ 下才有，大部分 IDE 都支持这两种编译模式。

实现方法

现在我们将创建第一个工程。通过选择 File| New Project |Project... 菜单，你可以创建不同的工程类型。我们从最简单的 Win32 Console Application 开始，如图 1.2 所示。

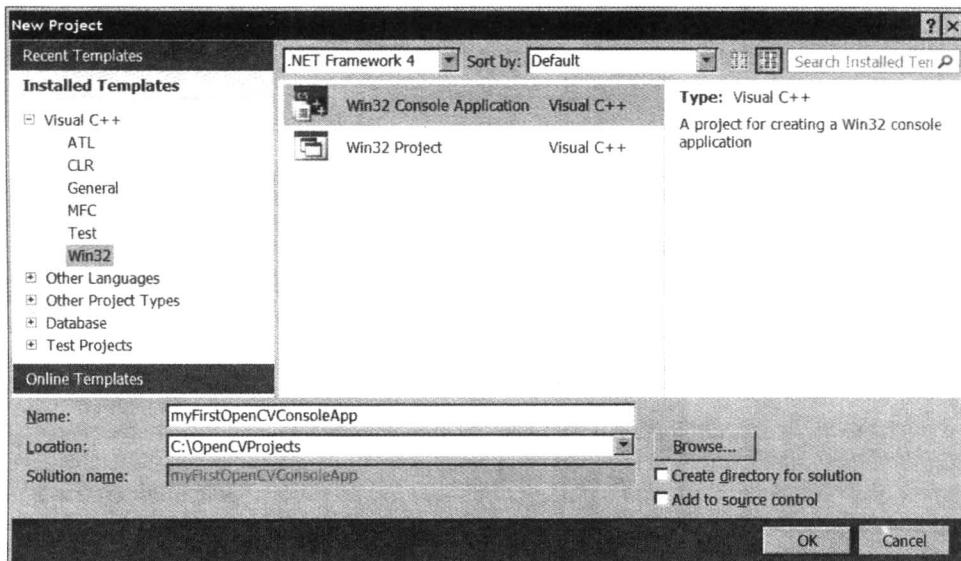


图 1.2

你需要指定项目创建的位置以及项目的名称，你也可以选择为解决方案创建一个文件夹（右下角）。如果选中，那么会创建一个额外的文件夹包含着解决方案。如果你简单地将它留着，那么扩展名为 .sln 的解决方案也会创建，但是它将与工程文件位于同一个文件夹。点击 OK 与 Next，进入 Win32 Application Wizard 的 Application Settings 窗口。正如图 1.3 所示，有一系列选项供选择，但是我们仅创建一个空的项目。

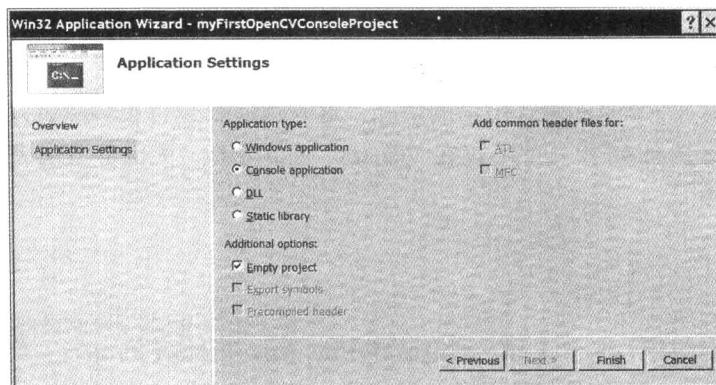


图 1.3

需要注意的是，我们取消了 Precompiled Header 选项，这是 Visual Studio 的预编译头文件特性，可以加速编译过程。由于我们希望遵守 ANSI C++ 标准，因此并不会开启该选项。这时点击 Finish，那么你的工程就创建完毕。它暂时是空白的，但是我们很快将添加一个主文件。

首先，为了能够编译及运行你未来的 OpenCV 应用，Visual C++ 必须知道 OpenCV 头文件以及库的位置。由于你今后将创建多个 OpenCV 工程，最好的方法是创建一个属性单供今后的各种项目使用。我们使用 Property Manager 来完成这项操作，如果你没有看到图 1.4 所示的界面，那么可以从 View 菜单访问它。

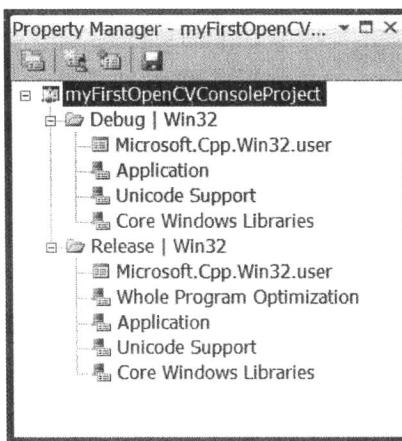


图 1.4

在 Visual C++ 2010 中，属性单是一个描述工程设置的 XML 文件。我们通过右键点击工程的 Debug | Win32 节点并选中 Add New Property Sheet 项可以新建一个属性单，

如图 1.5 所示。

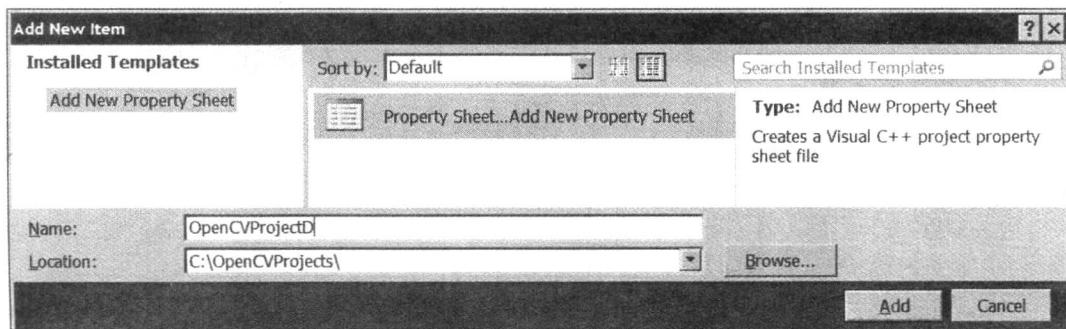


图 1.5

点击 Add 后新建的属性单便被加入，之后我们需要进行编辑。双击属性单的名称并选择 VC++ Directories，如图 1.6 所示。

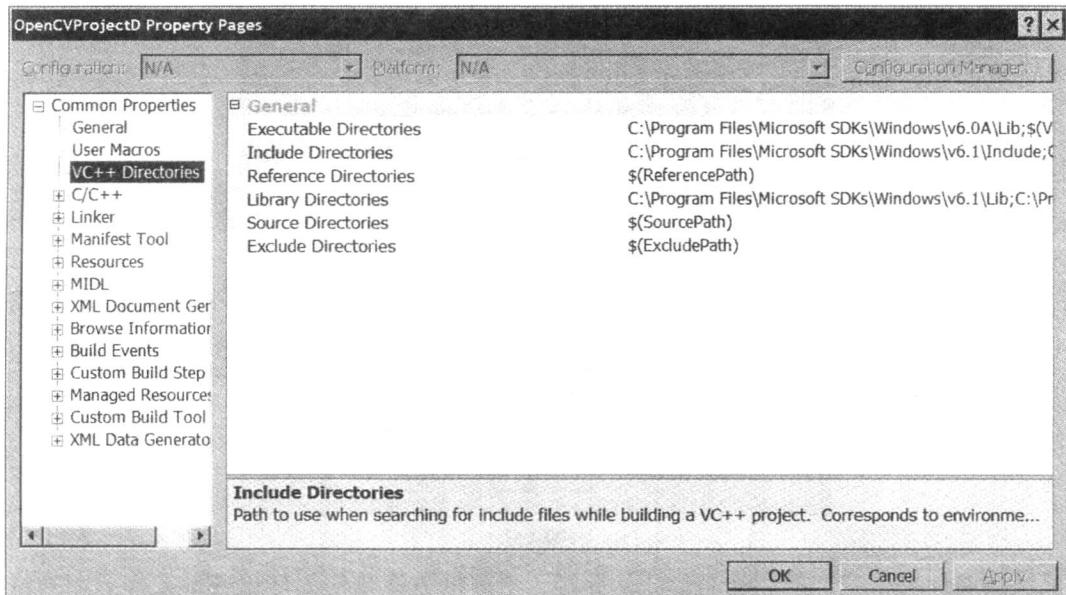


图 1.6

编辑 Include Directories 文本框，并且添加 OpenCV 的头文件的路径，如图 1.7 所示。

对 Library Directories 进行同样的操作，这次添加的是 OpenCV 的库文件的路径，如图 1.8 所示。

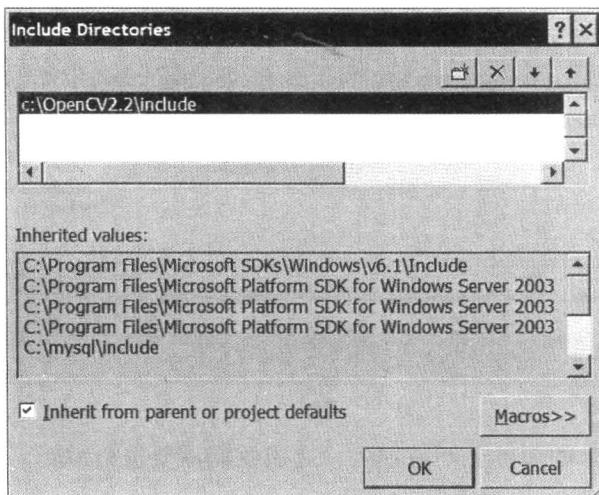


图 1.7

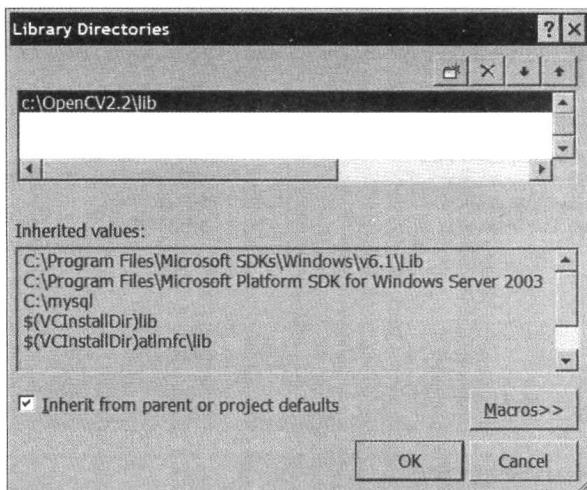


图 1.8

在属性单中我们使用的是显式的路径，更好的方案是使用环境变量来标示库的位置。这样，如果我们切换至另一个版本，只需要简单修改这个变量令它指向库的新位置。同时，在团队开发的项目中，不同的用户可能将库安装在不同的位置。使用环境变量即可避免不同用户修改属性单。因此，如果你定义环境变量 OPENCV2_DIR 为 c:\OpenCV2.2，那么属性单中的两个路径将分别为 \$(OPENCV_DIR)\include 和 \$(OPENCV_DIR)\lib。

下一步是指定需要链接的 OpenCV 库文件，不同的应用程序所需的 OpenCV 模块也可能不同。由于我们需要在所有的项目中重用该属性单，我们将添加书中所需的库模块。进入 Linker 节点的 Input 项，如图 1.9 所示。

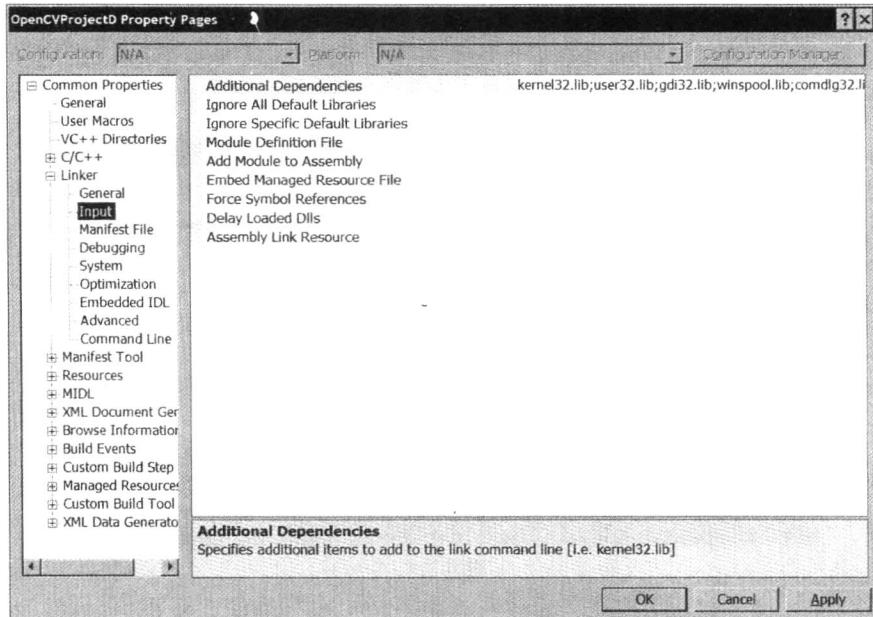


图 1.9

编辑 Additional Dependencies 文本框，添加图 1.10 所示的库模块。

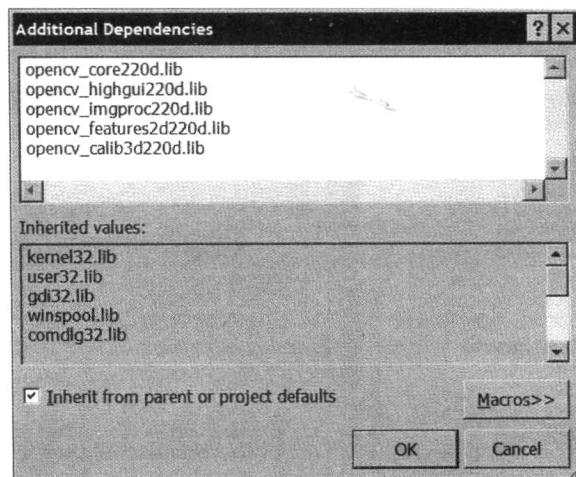


图 1.10

我们指定的库都以字母“d”作为结尾，这表示它们是 Debug 模式下的二进制文件。你需要为 Release 模式创建另一个（几乎相同的）属性单，将它添加在 Release|Win32 节点下。这一次，库文件的名称将不包含字母“d”。

现在我们准备好创建、编译并运行首个应用程序。首先使用 Solution Explorer 添加一个新的源文件，并在 Source Files 节点上右击。选中 Add New Item，指定该 C++ 文件的名称为 main.cpp，如图 1.11 所示。

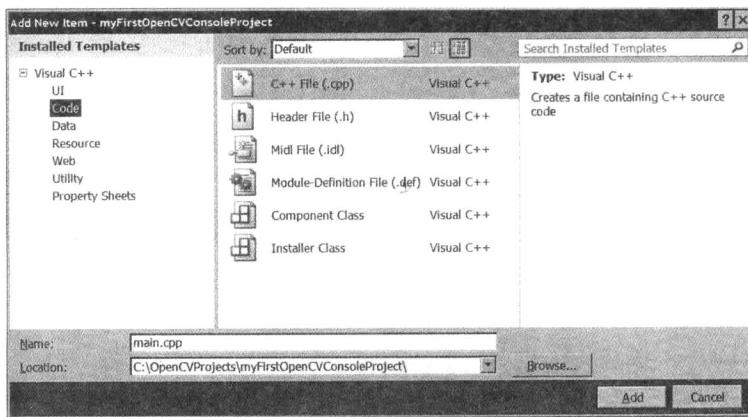


图 1.11

你也可以使用 File|New|File... 菜单完成相同的工作。现在，让我们创建一个简单的应用，它将显示默认文件夹中名为 img.jpg 的图像，如图 1.12 所示。

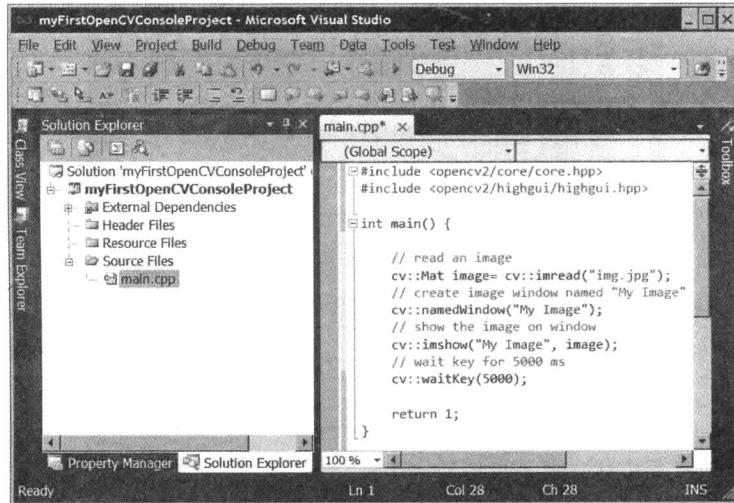


图 1.12

一旦复制好了代码（我们将在稍后进行解释），你就可以编译并使用屏幕顶部工具栏中的 Start 绿色箭头运行它了。图像将在屏幕上显示 5s，如图 1.13 所示。

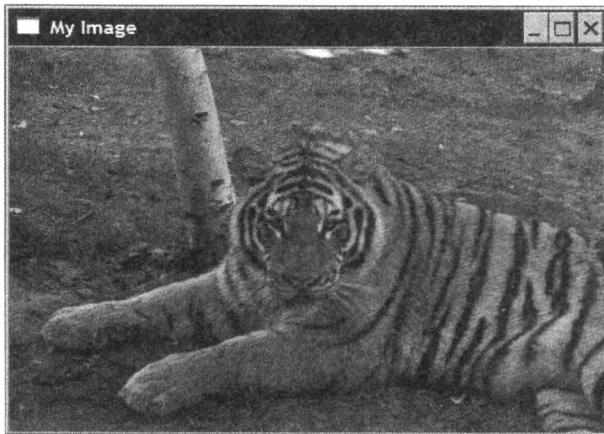


图 1.13

如果一切顺利，那么你已经完成了你的首个成功的 OpenCV 应用！如果程序运行时出错，那么很有可能是无法找到图像文件。下一部分将指导你如何将图像文件放入正确的文件夹。

作用原理

点击 Start Debugging 按钮（或按 F5），便开始编译项目，然后执行它。你也可以点击 Build 菜单下的 Build Solution（或是按 F7）来进行编译。当你首次编译项目时，将会生成一个 Debug 文件夹，它包含可执行文件（后缀为 .exe）。类似地，你也可以在下拉框中选择 Release 配置以生成 Release 版本的可执行文件，它位于绿色箭头按钮的右侧。你也可以使用 Build 菜单中的 Configuration Manager，此时创建的是 Release 文件夹。

当你使用 Start 按钮执行一个项目时，默认的文件夹将总是包含解决方案的文件夹。然而，如果你在 IDE 之外通过双击执行应用程序（从资源管理器中），那么默认文件夹将是包含可执行文件的文件夹。因此，在运行前需确保图像文件位于合适的文件夹中。

参 考

本章后续的“载入、显示及保存图像”秘诀解释了本节中的源代码。

1.4 使用Qt创建OpenCV项目

Qt 是一个完整的 C++ 集成开发环境（IDE），最初由挪威软件公司 Trolltech 开发，在 2008 年时被 Nokia 收购。它提供两套协议——开源的 LGPL 协议，以及用于商业开发的付费协议。它由两个独立元素组成——跨平台的 Qt Creator 开发环境，以及 Qt 类库与开发工具。使用 Qt 软件开发包（SDK）进行 C++ 应用开发的优点如下。

- ◆ 它是由 Qt 社区带来的开源项目，使你能够访问不同的 Qt 组件的代码。
- ◆ 它是跨平台的，这意味着你开发的应用可以运行在不同的操作系统，包括 Windows、Linux 及 Mac OS X。
- ◆ 它包含一个完整的跨平台 GUI 库，遵循高效的面向对象以及事件驱动模型。
- ◆ Qt 同时包括多媒体、图形、数据库、多线程、网页应用等方面的跨平台库，可用于开发高级应用。

准备工作

Qt 可以从 <http://qt.nokia.com> 下载。如果你选择的是 LGPL，那么它是免费的。你应该下载完整的 SDK，同时确保选择合适平台的库文件。当然，由于我们使用的是开源软件，总是可以重新进行编译。

本书中使用的是 1.2.1 版本的 Qt Creator，而 Qt 的版本是 4.6.3。需要注意的是，在 Qt Creator 的项目标签界面中能够管理你安装的不同版本 Qt。这确保你总是能使用合适的 Qt 进行项目编译。

实现方法

当你运行 Qt Creator 时，它会问你如果你需要创建新项目，或者打开最近的一个。您也可以在 File 菜单的 New 选项中创建一个新的项目。为了重现我们在前一则秘诀中的结果，我们选择创建 Qt4 控制台应用程序，如图 1.14 所示。

之后，你可以指定名称及项目所在位置，如图 1.15 所示。

下面的屏幕将允许您选择需要包含的模块。此处我们保持默认的选项不变，并一路单击 Next 与 Finish。一个空的控制台应用便创建了，如图 1.16 所示。

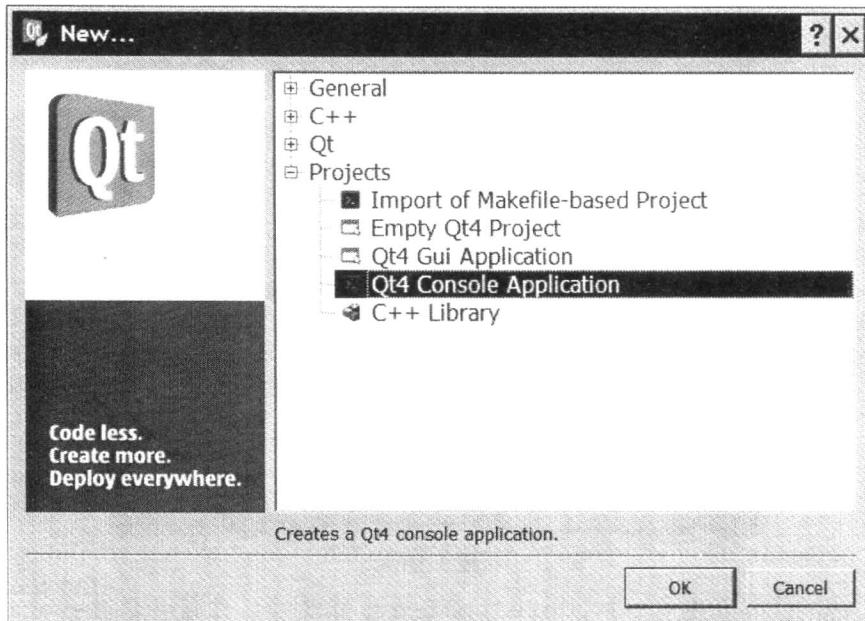


图 1.14

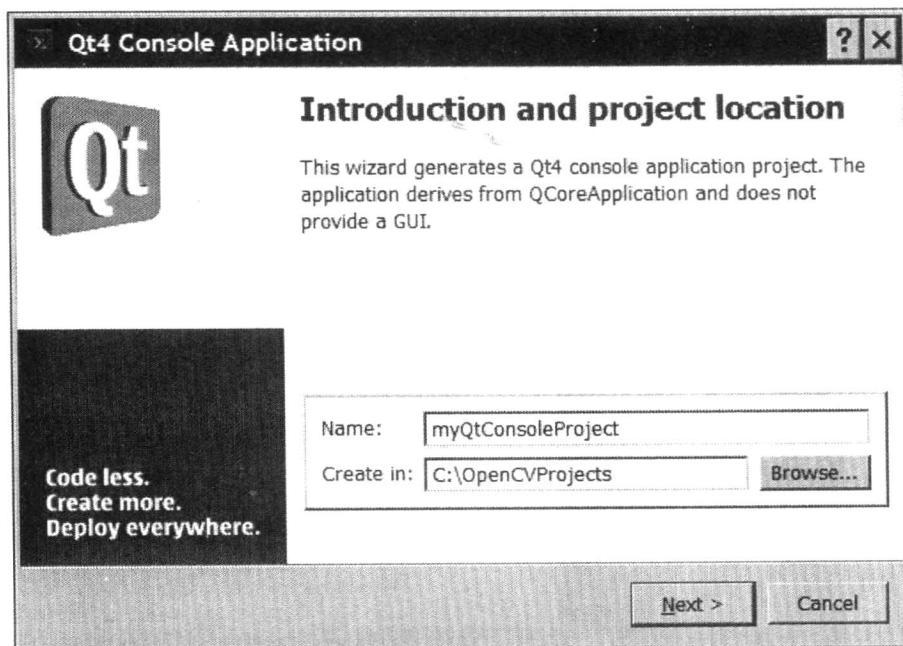


图 1.15

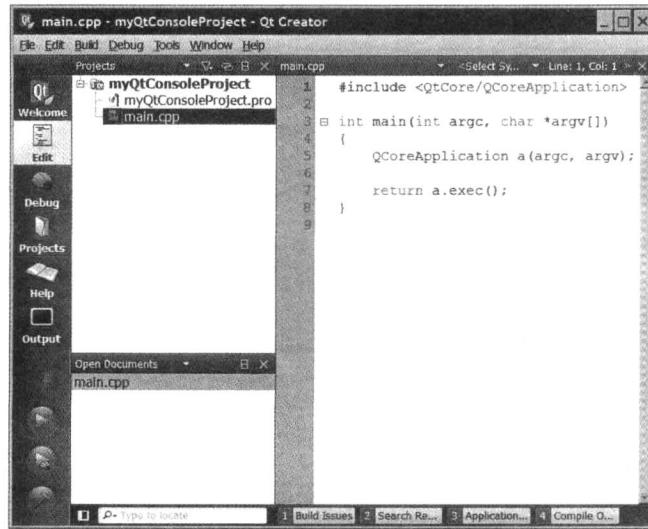


图 1.16

Qt 生成的代码创建了一个 `QCoreApplication` 对象，并调用它的 `exec()` 方法。这只有在应用程序需要事件处理器来负责用户与 GUI 交互时才是必需的。在我们简单地打开并显示图像的例子中，这是没有必要的。在将生成的代码替换为之前使用的范例后，这个简单的显示图像的程序将拥有图 1.17 所示的代码。

A screenshot of the Qt Creator IDE showing the same project structure as in Figure 1.16. The main editor window now contains the following code, which includes OpenCV headers and logic to read an image and display it in a window:

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
int main() {
    // read an image
    cv::Mat image= cv::imread("img.jpg");
    // create image window named "My Image"
    cv::namedWindow("My Image");
    // show the image on window
    cv::imshow("My Image", image);
    // wait key for 5000 ms
    cv::waitKey(5000);

    return 1;
}
```

The status bar at the bottom shows tabs for "Build Issues", "Search Results", "Application Output", and "Compile Output".

图 1.17

为了能够编译程序，OpenCV 库文件和头文件的路径必需指定。在 Qt 中，这些信息包含在后缀为 .pro 的项目文件中，这是一个简单的描述项目参数的文本文件。你可以在 Qt Creator 中编辑这个项目文件，如图 1.18 所示选中对应的项目文件。

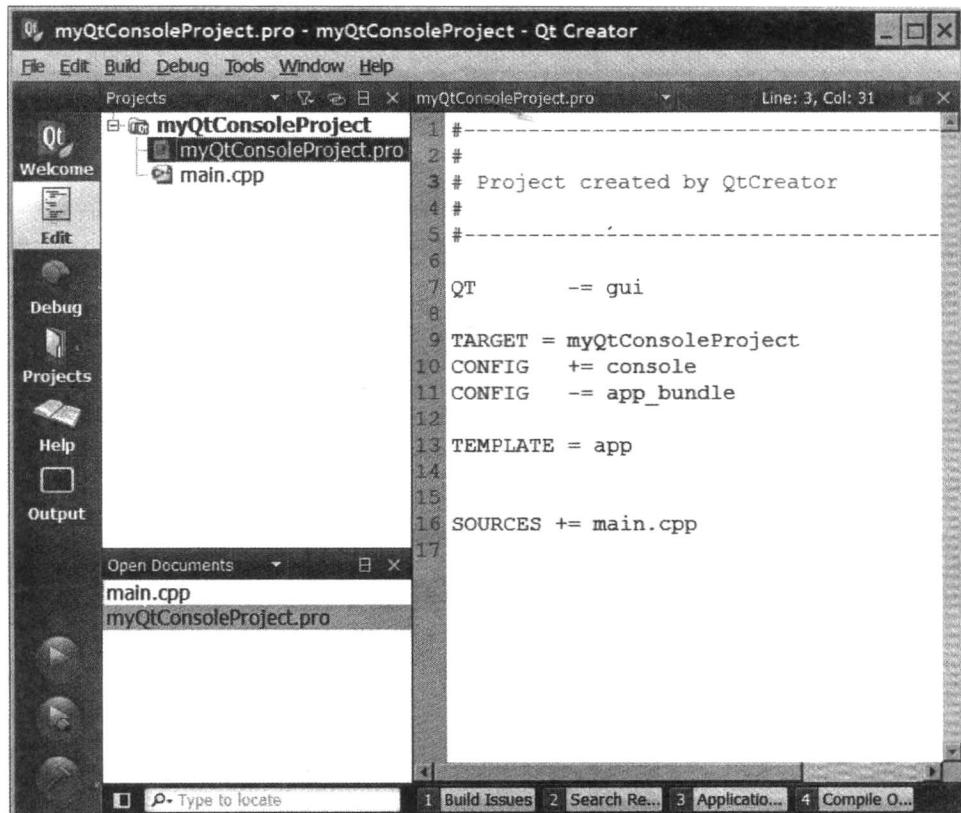


图 1.18

用于构建 OpenCV 应用的信息可以添加在项目文件的末尾：

```
INCLUDEPATH += C:\OpenCV2.2\include\
LIBS    += -LC:\OpenCV2.2\lib\
-lopencv_core220\
-lopencv_highgui220\
-lopencv_imgproc220\
-lopencv_features2d220\
-lopencv_calib3d220
```

下载范例



在 <http://www.packtpub.com> 购买了 Packt 书籍后，你可以使用购买账号下载书中的范例。

如果你是在别处购买的书籍，那么可以访问 <http://www.packtpub.com/support> 进行注册，文件将以电子邮件形式发送给你。

现在程序可以进行编译和执行了，通过点击底部左侧的绿色箭头或是按 Ctrl+R 即可完成。使用 Projects 标签下的 Build Settings 即可设置 Debug 模式或 Release 模式，如图 1.19 所示。

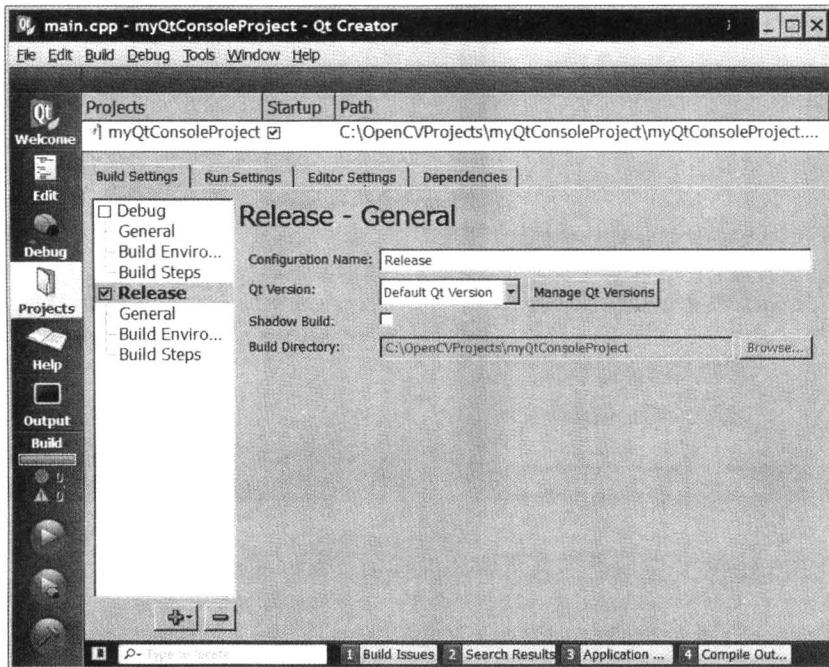


图 1.19

作用原理

项目文件描述了一个 Qt 项目，它是一个声明了许多变量的文本文件，包含用于构建项目的相关信息。当处于编译过程时，这个文件将会被软件工具 qmake 调用。文件

中的每个变量都与一系列的值相联系。qmake 可以识别的主要变量如下：

- ◆ TEMPLATE：定义项目的类型（应用程序、库，等等）。
 - ◆ CONFIG：指定编译器的不同选项。
 - ◆ HEADERS：列出项目的头文件。
 - ◆ SOURCES：列出项目的源文件（.cpp）。
 - ◆ QT：声明所需的 Qt 扩展模块及类。默认包含核心模块及 GUI 模块。如果你希望移除它们其中之一，可以使用 -= 语句。
 - ◆ INCLUDEPATH：指定用于搜索头文件的文件夹。
 - ◆ LIBS：包含链接时的库文件列表。使用 -L 指定文件夹路径，-l 指定库的名称。
- 以上便是最常用的变量。

扩展阅读

项目文件中可以使用许多额外的特性，如可以针对每个平台定义不同的变量：

```
win32{
    #Windows 32 平台专属的变量
}
unix{
    #Unix 32 平台专属的变量
}
```

你也可以使用 pkg-config 工具包，这个开源的工具用于辅助配置正确的编辑器选项和库文件。当你使用 CMake 安装 OpenCV 时，unix-install 包含一个 opencv.pc 文件，pkg-config 读取该文件来决定编译时的参数。此时，跨平台的 qmake 项目文件变成：

```
unix{
    CONFIG += link_pkgconfig
    PKGCONFIG += opencv
}
Win32{
INCLUDEPATH += C:\OpenCV2.2\include\
LIBS += -LC:\OpenCV2.2\lib\
    -lopencv_core220\
        -lopencv_highgui220\
        -lopencv_imgproc220\
```

```
-lopencv_features2d220\  
-lopencv_calib3d220  
}
```

参 考

下一则秘诀“载入、显示及保存图像”解释了本节中用到的 OpenCV 源代码。

访问 <http://qt.nokia.com> 可以查询 Qt、Qt Creator 以及所有 Qt 扩展模块的完整文档。

1.5 载入、显示及保存图像

先前的两则秘诀介绍了如何创建简单的项目，但是我们还没有解释使用到的代码。本则秘诀将展示如何执行最基础的操作，包括从外部文件载入图像、在窗口中显示图像以及存储图像到磁盘上。

准备工作

使用 Visual Studio 或是 Qt Creator，新建一个控制台程序，填充 main 函数的过程可以参考先前的两则秘诀。

实现方法

第一件事是声明一个表示图像的变量，在OpenCV 2中，这个变量将是 cv::Mat 类型。

```
cv::Mat image;
```

这句话将创建宽高都为 0 的图像，通过调用 cv::Mat 的 size() 方法可以获取该图像的尺寸，我们可以由此进行验证。该方法的返回值是一个结构体，包含着宽度和高度：

```
std::cout<<"size,"<<image.size().height<<","  
<<image.size().width<<std::endl;
```

接着，一个简单的读取函数将进行文件读取、解码以及内存的分配。

```
image= cv::imread("img.jpg");
```

在使用该图像前需要先检查图像已经被正确读取，如果文件不存在、文件损坏或者格式无法识别，将会出现错误。检验的方法如下：

```
if (!image.data){  
    // 图像尚未创建.....  
}
```

此处的成员变量 `data` 事实上是指向已分配的内存块的指针，包括图像数据。当不存在数据时，它被简单设置为 0。你也许立刻就想显示这幅图像，OpenCV 的 `highgui` 模块实现了该功能。首先定义一个需要进行图像显示的窗口，接着指定需要显示的图像：

```
cv::namedWindow("Original Image");// 定义窗口  
cv::imshow("Original Image",image);// 显示图像
```

现在如果你希望对图像进行一些处理，OpenCV 也提供了很多处理函数，本书将介绍其中一些。首先介绍将图像水平反转的函数。OpenCV 中的多个图像变换可以在原地进行，即不创建新的图像，直接修改输入图像的内容。`flip` 函数便是其中一种，然而，我们也可以创建另外一个矩阵来保存输出结果：

```
cv::Mat result;  
cv::flip(image,result,1);// 整理表示水平反转  
// 0 表示垂直反转  
// 负数表示既有水平也有垂直反转
```

结果显示在另一个窗口中：

```
cv::namedWindow("Output Image");  
cv::imshow("Output Image",result);
```

由于这是个控制台窗口，它会在 `main` 函数的结尾处终止，因此我们添加额外的 `highgui` 函数使得程序退出前不停等待用户的按键输入：

```
cv::waitKey(0);
```

你可以在两个不同的窗口中看到输入图像和输出图像。最后，你也许希望将处理后的图像保存在磁盘上。这通过下面的 `highgui` 函数得到实现：

```
cv::imwrite("output.bmp", result);
```

文件的后缀名决定了图像保存时的编码格式。

作用原理

OpenCV 中的 C++ 类和函数都定义在命名空间 cv 之内，有两种方法可以访问它们。在 main 函数前加入下述语句：

```
using namespace cv;
```

或者在使用 OpenCV 类和函数时在名字前加入 cv:: 命名空间，这也正是我们在本则秘诀中所做的。

cv::Mat 类是用于保存图像以及其他矩阵数据的数据结构。默认情况下，它们的尺寸为 0，但是你也可以指定初始尺寸：

```
cv::Mat ima(240, 320, CV_8U, cv::Scalar(100));
```

同时，你还需要指定矩阵中元素的类型，这里的 CV_8U 对应的是单字节的像素图像。字母 U 意味着无符号的（ Unsigned ）。你也可以使用字母 S 声明带符号的（ Signed ）类型。对于彩色图像，你需要指定 3 个通道（ CV_8UC3 ）。你也可以声明 16 位或 32 位的（无符号的或带符号的）整数图像，如 CV_16SC3 。你也可以声明 32 位或 64 位的浮点数，如 CV_32F 。

当 cv::Mat 对象离开作用域后，分配的内存将自动释放。这很方便，可以避免内存泄露的困扰。另外， cv::Mat 实现了引用计数以及浅拷贝，当图像之间进行赋值时，图像数据并没有发生复制，两个对象都指向同一块内存块。这也可用于参数传值的图像，以及返回值传值的图像。引用计数的作用是当只有当所有引用内存数据的对象都被析构后，才会释放内存块。如果你希望创建的图像拥有原始图像的崭新拷贝，那么可以使用 copyTo() 方法。在范例中，我们声明额外的图像来测试这个行为，代码如下：

```
cv::Mat image2, image3;
image2= result; // 两个图拥有同一份数据
result.copyTo(image3); // 创建新的拷贝
```

现在如果你再次翻转输出图像，并显示两幅额外的图像，那么你将看到 image2 也受到变换的影响（因为它指向的内存数据与输出图像是同一份），而 image3 保持

原样因为它包含的是一份新的拷贝。`cv::Mat` 的内存分配的模型也意味着你自己成功地编写返回图像的函数或类的方法：

```
cv::Mat function() {
    // 创建图像
    cv::Mat ima(240,320,CV_8U, cv::Scalar(100));
    // 并返回它
    return ima;
}
```

如果我们从 `main` 函数中调用该函数：

```
// 得到灰度图
cv::Mat gray= function();
```

`gray` 变量将包含函数中创建的图像内容，而不涉及额外的内存分配。事实上，函数返回的 `cv::Mat` 对象在转移到 `gray` 时仅发生浅拷贝。函数中的声明局部变量 `ima` 离开作用域时被析构，但是由于它所关联的引用计数表示内部图像正被另一个对象 `gray` 所引用，因此内存块并不会被释放。

然而，在类中使用时你需要谨慎，避免返回类中的图像成员。以下便是一个容易引起错误的例子：

```
class Test{
    // 图像成员变量
    cv::Mat ima;
public:
    // 构造函数，创建灰度图
    Test() :ima(240,320,CV_8U, cv::Scalar(100)) {}
    //method 返回一个成员变量，这是个坏主意 .....
    cv::Mat method() {return ima;}
};
```

如果某个函数调用了该方法，那么它便得到该图像的浅拷贝。如果之后这份拷贝发生改变，那么类中的成员变量也会一同被修改，这将影响之后的行为（反之亦然）。为了避免这种错误的发生，你应该返回成员变量的一份新的拷贝。

扩展阅读

OpenCV 2 引入了崭新的 C++ 接口。之前使用的 C 函数和数据结构仍然可以使用，其中图像是通过 `IplImage` 进行操作的，该结构继承自 IPL 库，即英特尔图像处理库（Intel Image Processing Library），现在已整合进 IPP 库，即英特尔综合性能库（Intel Integrated Performance Primitive Library）。如果你使用老式的 C 风格接口，那么你需要操作 `IplImage` 结构。幸运的是，可以方便地转换一个 `IplImage` 到 `cv::Mat` 对象。

```
IplImage* iplImage = cvLoadImage("c:\\\\img.jpg");
cv::Mat image4(iplImage, false);
```

`cvLoadImage` 是一个 C 风格的图像读取函数。`cv::Mat` 对象构造函数中的第二个参数说明不需要进行数据拷贝（设为 `true` 意味着得到崭新的拷贝，默认值为 `false`），即 `IplImage` 与 `image4` 共享同一份图像数据。此处你需要谨慎，以免创建野指针。由于这个原因，OpenCV 2 还提供了一个实现引用计数的指针类，我们可以使用它来封装 `IplImage` 指针：

```
cv::Ptr<IplImage> iplImage = cvLoadImage("c:\\\\img.jpg");
```

否则，当你需要释放 `IplImage` 结构指向的内存时，必须明确地调用：

```
cvReleaseImage(&iplImage);
```

你应该尽量避免使用这个被废弃的数据结构，而是使用 `cv::Mat`。

1.6 使用Qt创建GUI应用

Qt 提供了丰富的软件库以构建复杂的 GUI，并使它拥有专业的外观。在 Qt Creator 的帮助下，制作 GUI 变得十分简便。本则秘诀将向你展示如何构建基于 Qt 的 OpenCV 应用，用户能够在软件中使用 GUI 进行控制。

准备工作

首先启动 Qt Creator，我们将用它来创建我们的 GUI 应用。不使用该软件也可以创建 GUI，但是使用可视化的 IDE 对控件进行拖拽是最为方便的方法。

实现方法

选择 Create New Project, 如图 1.20 所示, 选择 Qt4 GUI Application。

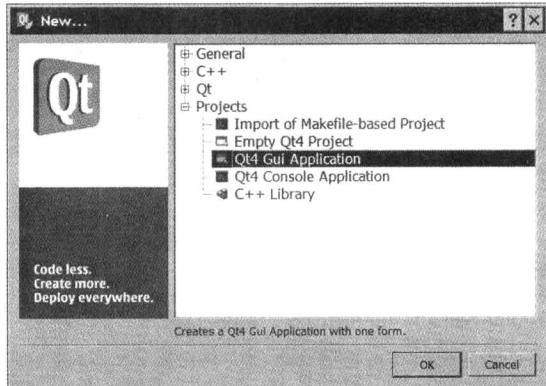


图 1.20

输入项目的名称与路径, 如果你点击 Next, 那么你将看到 Qt GUI Module 已经被选中。由于我们暂时不需要其他模块, 你可以立即点击 Finish。这将创建新的项目, 除了通常的项目文件 (.pro) 以及 main.cpp 文件, 你还将看到两个 mainwindow 文件定义 GUI 窗口的内容。你将看到 .ui 后缀的文件, 它定义了 UI 的布局。事实上, 如果你双击它, 将看到当前的用户界面, 如图 1.21 所示。

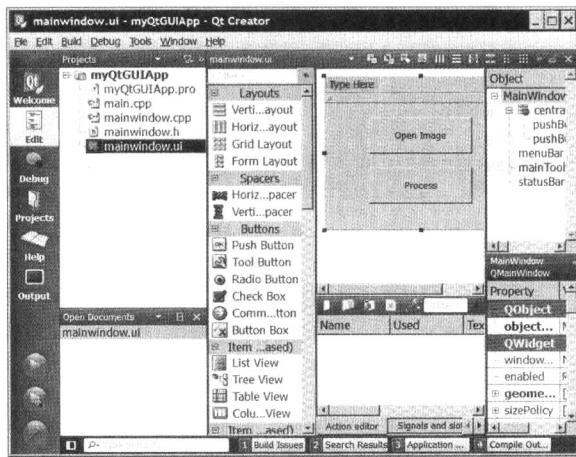


图 1.21

你能在界面上拖拽不同的控件, 正如前例中所做的我们将两个按钮拖入界面中。你可以改变按钮以及窗口的尺寸, 也可以重命名按钮的标签。只需要点击文字, 并输

入你想要的名称即可。

现在，让我们加入一个信号函数以处理点击按钮的事件。右键单击第一个按钮，在上下文菜单中选择 Go to slot，可选的信号将如图 1.22 所示。

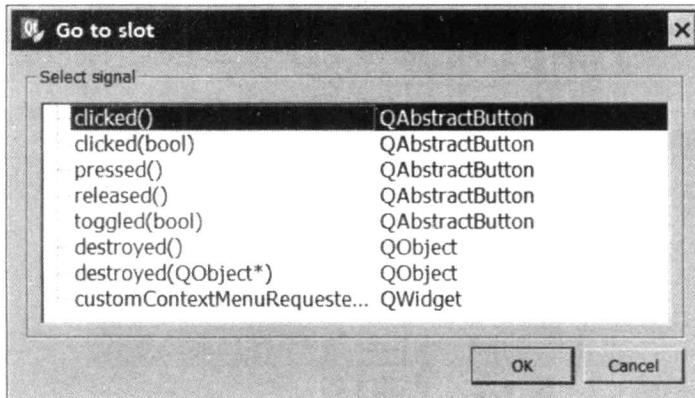


图 1.22

此处我们选择 clicked() 信号，它处理的是按钮单击的事件。之后我们将进入到mainwindow.cpp 文件，你将看到文件中添加了新的函数。这是一个槽(Slot)函数，当接收到 click() 信号时它将被调用：

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}
MainWindow::~MainWindow()
{
    delete ui;
}
void MainWindow::on_pushButton_clicked()
```

为了能够显示并处理图像，我们需要定义一个 cv::Mat 成员变量。我们对 MainWindow 类进行修改，现在的头文件是

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include<QtGui/QMainWindow>
#include<QFileDialog>
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
namespace Ui
{
    class MainWindow;
}
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    Ui::MainWindow*ui;
    cv::Mat image;// 图像变量
private slots:
    void on_pushButton_clicked();
};
#endif // MAINWINDOW_H
```

注意，我们还引入了 core.hpp 和 highgui.hpp 头文件，为了正常编译，还需要编辑项目文件加入 OpenCV 库的信息。

OpenCV 的代码将开始添加进项目，第一个按钮打开原始图像。我们将下述代码加入对应的槽函数中：

```
void MainWindow::on_pushButton_clicked()
{
    QString fileName = QFileDialog::getOpenFileName(this,
        tr("Open Image"), ".",
        tr("Image Files (*.png *.jpg *.jpeg *.bmp)"));
    image=cv::imread(fileName.toAscii().data());
    cv::namedWindow("Original Image");
    cv::imshow("Original Image", image);
}
```

接着右键单击第二个按钮，并添加一个新的槽函数，它将对选中的输入函数进行操作。下述代码用于进行简单的图像翻转：

```
void MainWindow::on_pushButton_2_clicked()
{
    cv::flip(image,image,1);
    cv::namedWindow("Output Image");
    cv::imshow("Output Image", image);
}
```

编译并运行这段代码后，你将得到带有两个按钮（图 1.23）的程序，实现了图像的选取和处理。

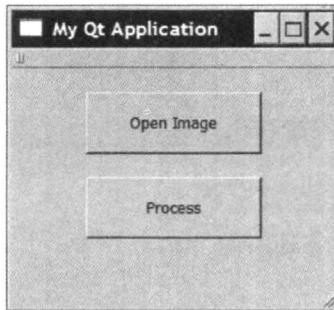


图 1.23

输入图像与输出图像显示在我们定义的两个 `highgui` 窗口中。

作用原理

在 Qt 的 GUI 编程框架中，对象之间通过信号与槽进行通信。当控件改变状态或是有事件发生时，便会发出信号。该信号有着预定义的签名，另外一个对象必须定义相同签名的槽才可以接收该信号。槽是一种特殊的类函数，当与它相关联的信号被发出时它将被自动调用。

信号与槽都被定义为类函数，但是必须在 Qt 的槽及信号名称下进行声明。这也正是 Qt Creator 在增加一个槽到按钮时所做的工作，即

```
private slots:
    void on_pushButton_clicked();
```

信号与槽是松散耦合的，即信号不需要了解任何拥有对应槽的对象，而槽也不需要指定是否有信号与之相连。同时，多个槽可以与一个信号相连，而一个槽也可以接

收来自多个对象的信号。唯一的需求是信号函数与槽函数的签名必须一致。

所有继承自 `QObject` 的类都含有信号与槽。它们通常是控件类 `QWidget` 的子类，但是其他的类也可以定义信号与槽。信号与槽是一种强大的类间通信机制，然而它仅局限于 Qt 框架中使用。

在 Qt 中，主窗口是 `MainWindow` 类的一个实例。你可以通过 `MainWindow` 类定义中的成员变量 `ui` 访问它。另外，图形界面中的每个控件都是一个对象。当 GUI 创建时，指向主窗口中每个控件的指针都与 `ui` 变量进行关联。因此，你可以访问程序中每个图形控件的属性与方法。例如，如果你希望在输入图像被选中前都禁用 `Process` 按钮，你所要做的是当 GUI 初始化时（调用 `MainWindow` 构造函数时）调用下述方法：

```
ui->pushButton_2->setEnabled(false);
```

指针变量 `pushbutton_2` 对应的是 `Process` 按钮。之后，当图像被成功载入后，你可以启用该按钮（`Open Image`）：

```
if (image.data) {  
    ui->pushButton_2->setEnabled(true);  
}
```

在 Qt 中，GUI 的布局被完整地描述于一个扩展名为 `.ui` 的 XML 文件中。如果你使用文本编辑器打开项目文件夹中的 `.ui` 文件，将看到这个文件的 XML 内容。文件中定义了多个 XML 标签，以本秘诀中展示的应用为例，你将看到两个定义为 `QPushButton` 的控件类的标签。这些控件类的标签中有一项为名称，对应的是与 `ui` 对象相关联的指针的名称。这些控件都定义了一个几何属性，描述它们的位置和尺寸。此外还有许多其他的属性。Qt Creator 有一个属性标签，展示每个控件的属性的值。因此，即使 Qt Creator 是最佳的 GUI 制作工具，你依旧可以编辑 `.ui` 文件来改变你的布局。

扩展阅读

在 Qt 中直接在界面中展示图像非常简单。你所需做的是添加一个标签（`label`）对象，然后将一幅图像赋予该标签以进行显示。你可以通过 `ui` 对象中的相应指针访问该标签对象，本例中即是 `ui->label`。但是该图像的类型必须是 `QImage`，Qt 中用于处理图像的数据结构。格式的转换相对简单，除了三个颜色的通道需要从 `cv::Mat` 的 BGR 顺序翻转为 `QImage` 中的 RGB 顺序。在使用 `cv::cvtColor` 函数实现该工作后，

我们的 Process 按钮的处理函数变为

```
void MainWindow::on_pushButton_2_clicked()
{
    cv::flip(image,image,1); // 改变图像
    // 改变彩色通道的顺序
    cv::cvtColor(image,image,CV_BGR2RGB);
    //Qt 图像
    QImage img= QImage((const unsigned char*)(image.data),
                       image.cols,image.rows,QImage::Format_RGB888);
    // 显示在 label 中
    ui->label->setPixmap(QPixmap::fromImage(img));
    // 改变 label 的尺寸以自适应图像
    ui->label->resize(ui->label->pixmap()->size());
}
```

最后，输出图像直接显示在 GUI 中，如图 1.24 所示。

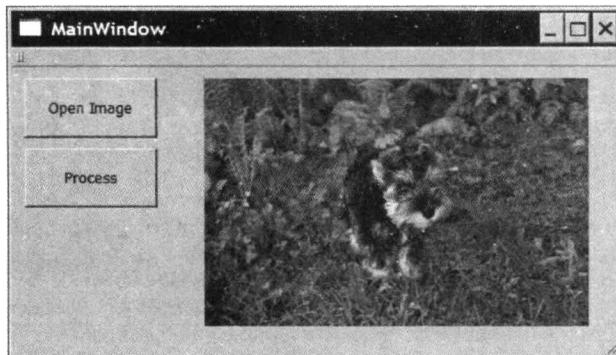


图 1.24

参 考

查询 Qt 位于 <http://qt-project.org/doc/> 的在线文档可以了解更多关于 Qt GUI 模块以及信号槽机制的信息。

第 2 章

操作像素

本章主要探讨：

- ◆ 存取像素值；
- ◆ 使用指针遍历图像；
- ◆ 使用迭代器遍历图像；
- ◆ 编写高效的图像遍历循环；
- ◆ 遍历图像和邻域操作；
- ◆ 进行简单的图像算术；
- ◆ 定义感兴趣区域。

2.1 引言

为了编写计算机视觉应用，你必须会存取图像的内容，如修改或者创建图像。本章将教会你如何操作图像的基本元素，即所谓的像素。你将学会如何遍历一张图像并且处理其像素。另外，还将学会高效的处理方法，因为即使普通的图像都可能包含数以万计的像素。从根本上来说，一张图像是一个由数值组成的矩阵。这也是 OpenCV 2 用 `cv::Mat` 这个数据结构来表示图像的原因。矩阵的每一个元素代表一个像素。对于灰度图像（仅包含“灰色”的图像）而言，像素由 8 位无符号数来表示，其中 0 代表黑色，255 代表白色。对于彩色图像而言，每个像素需要三个这样的 8 位无符号数来表示三个颜色通道（红、绿、蓝）。因此，在这种情况下，矩阵的元素是一个三元数。

我们在前一章中看到，OpenCV 允许我们创建不同像素类型的矩阵或图像。例如，整型 (`CV_8U`) 或者浮点型 (`CV_32F`)，它们在一些图像处理过程中，用来保存诸如中间值这样的内容时非常有用。大多数矩阵运算可以被应用于任意类型的矩阵，但是有些运算对数据类型或矩阵的通道数有所要求。因此，对函数前提条件的良好理解是避免出现常见的编程错误的必要条件。本章我们将使用图 2.1 作为测试图像（本书网站

上可以看到彩色图像)。

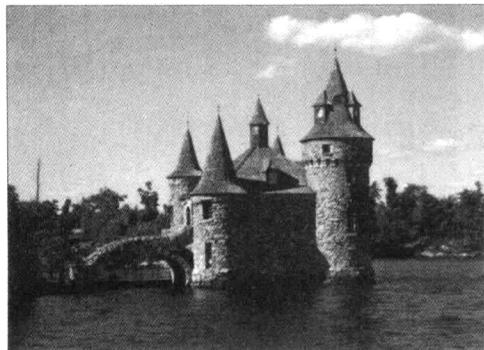


图 2.1

2.2 存取像素值

为了存取矩阵元素，你需要在代码中指定元素所在的行和列。程序会返回相应的元素。如果图像是单通道的，返回值是单个数值；如果图像是多通道的，返回值则是一组向量（Vector）。

准备工作

我们通过一个简单的函数来演示如何直接存取像素值。该函数会在图像中加入椒盐噪点。椒盐噪点是一种特殊的噪声，顾名思义，它随机地将部分像素设置为白色或者黑色。在传输过程中，如果部分像素值丢失，那么这种噪声就会出现。在我们的例子中，我们随机挑选若干像素，并将其设置为白色。

实现方法

我们创建一个函数，它的第一个参数是一张输入图像，该函数会修改此图像。为达到这个目的，我们需要使用传引用的参数传递方式。这个函数的第二个参数是我们欲将其替换成白色像素点的像素点个数：

```
void salt(cv::Mat &image, int n) {  
    for(int k=0;k<n;k++) {  
        //rand() 是随机数生成函数  
    }  
}
```

```
//Qt 中可以使用 qrand()
int i= rand()%image.cols;
int j= rand()%image.rows;
if(image.channels() == 1){// 灰度图
    image.at<uchar>(j,i)=255;
} else if(image.channels() == 3){// 彩色图
    image.at<cv::Vec3b>(j,i)[0]= 255;
    image.at<cv::Vec3b>(j,i)[1]= 255;
    image.at<cv::Vec3b>(j,i)[2]= 255;
}
```

这个函数由一个单层循环构成。每次循环将一个随机选取的像素的值设置为 255。随机选取的像素的行号 *i* 和列号 *j* 是通过一个随机函数得到的。此处我们通过检查图像的通道数来区分灰度图像和彩色图像。对于灰度图像，我们可以直接将像素值设为 255；对于彩色图像，我们需要将每个通道的值都设置为 255 才能得到一个白色像素。

你可以先打开一个图像，然后将在调用这个函数时候将此图像传递给它。

```
// 打开图像
cv::Mat image= cv::imread("boldt.jpg");
// 调用函数增加噪点
salt(image,3000);
// 显示图像
cv::namedWindow("Image");
cv::imshow("Image",image);
```

处理后的结果，图像看起来如图 2.2 所示。



图 2.2

作用原理

类 `cv::Mat` 有若干成员函数可以获取图像的属性。公有成员变量 `cols` 和 `rows` 给出了图像的宽和高。成员函数 `at<int y, int x>` 可以用来存取图像元素。但是必须在编译期知道图像的数据类型，因为 `cv::Mat` 可以存放任意数据类型的元素。这也是这个函数用模板函数来实现的原因。这也意味着，当调用该函数时，你需要使用以下方式指定数据类型：

```
image.at<uchar>(j,i)= 255;
```

注意：程序员一定要确保指定的数据类型要和矩阵中的数据类型相符合。`at` 方法本身不会进行任何数据类型转换。

对于彩色图像，每个像素由三个部分构成：红色通道、绿色通道和蓝色通道。因此，一个包含彩色图像的 `cv::Mat` 会返回一个由三个 8 位数组成的向量。OpenCV 将此类向量定义为 `cv::Vec3b`，即由三个 `unsigned char` 组成的向量。这也解释了为什么存取彩色图像像素的代码可以写为以下形式：

```
image.at<cv::Vec3b>(j,i)[channel]= value;
```

其中，索引值 `channel` 标明了颜色通道号。

类似地，OpenCV 还有二元素向量类型和四元素向量类型 (`cv::Vec2b` 和 `cv::Vec4b`)。OpenCV 同样拥有针对其他数据类型的向量类型，如 `s` 代表 `short`，`i` 代表 `int`，`f` 代表 `float`，`d` 代表 `double`。所有的这些类型都是使用模板类 `cv::Vect<T,N>` 定义的，其中 `T` 代表类型，`N` 代表向量中的元素个数。

扩展阅读

有时候使用 `cv::Mat` 的成员函数会很麻烦，因为返回值的类型必须通过在调用时通过模板参数指定。因此，OpenCV 提供了类 `cv::Mat_`，它是 `cv::Mat` 的一个模板子类。在事先知道矩阵类型的情况下，使用 `cv::Mat_` 可以带来一些便利。这个类额外定义了一些方法，但是没有任何成员变量，所以此类的指针或者引用可以直接进行相互类型转换。该类重载了操作符 ()，允许我们可以通过它直接存取矩阵元素。因此，假设有一个 `uchar` 型的矩阵，我们可以这样写：

```
cv::Mat<uchar> im2= image;//im2 指向 image  
im2(50,100)= 0;//存取第 50 行,100 列
```

由于 `cv::Mat_<uchar>` 的元素类型在创建实例的时候已经声明，操作符 `()` 在编译期就知道需要返回的数据类型。使用操作符 `()` 得到的返回值和使用 `cv::Mat` 的 `at` 方法得到的返回值是完全一致的，而且写起来更加简洁。

参 考

“编写高效的图像遍历循环”秘诀中对于该函数的效率有所讨论。

2.3 使用指针遍历图像

在大多数的图像处理中，为了计算，我们需要遍历图像的所有像素。考虑到将要访问的像素个数非常之多，高效地遍历图像时非常重要的。接下来的两个方法展示了两种不同的图像遍历循环实现方法。第一种使用的是指针算术。

准备工作

我们通过一个简单的例子来说明如何遍历图像：减少图像中的颜色数目。

彩色图像由三个通道组成。每个通道对应三原色（红、绿、蓝）之一的强度。由于每个强度值都是用一个 8 位的 `unsigned char` 表示，所以全部可能的颜色数目为 $256 \times 256 \times 256$ ，大于 1600 万个。理所当然，为了降低分析的复杂度，降低图像中的颜色数目有时候是有用的。一个简单的办法就是将 RGB 空间划分为同等大小的格子。例如，将每个维度的颜色数降低为原来的 $1/8$ ，那么总的颜色数就为 $32 \times 32 \times 32$ 。原始图像中的每个颜色都替换为它所在格子的中心对应的颜色。因此，这个算法很简单：如果 N 是颜色缩小比例，那么对于图像中每个像素的每一个通道，将其值除以 N （整数除法，舍去余数），然后再乘以 N ，这样就能得到不大于原始像素值的 N 的最大倍值。如果对每个 8 位通道的值都进行上述操作，那么就可以得到共计 $256/N \times 256/N \times 256/N$ 个颜色值。

实现方法

我们的颜色缩减函数原型如下：

```
void colorReduce(cv::Mat &image, int div=64);
```

用户提供一个图像和缩减因子。处理过程是 In-place 的，意味着输入图像的像素值会被此函数修改。参考更多内容部分，那里有一个更通用的函数，它同时包含一个输入图像参数和一个输出图像参数。整个处理过程通过一个双重循环来遍历所有的像素值：

```
void colorReduce(cv::Mat&image, int div=64){  
int nl= image.rows;// 行数  
// 每行的元素个数  
int nc= image.cols*image.channels();  
for(int j=0;j<nl;j++){  
    // 得到第 j 行的首地址  
    uchar* data= image.ptr<uchar>(j);  
    for (int i=0;i<nc;i++){  
        // 处理每一个像素 -----  
        data[i]= data[i]/div*div+div/2;  
        // 像素处理完成 -----  
    } // 行处理完成  
}  
}
```

整个函数可以通过以下的代码片段测试：

```
// 装载图像  
image= cv::imread("boldt.jpg");  
// 处理图像  
colorReduce(image);  
// 显示图像  
cv::namedWindow("Image");  
cv::imshow("Image",image);
```

结果如图 2.3 所示（本书网站上可以查看彩色图像）。



图 2.3

作用原理

在一个彩色图像中，图像数据缓冲区中的前三个字节对应图像左上角像素的三个通道值，接下来的三个字节对应第一行的第二个像素，依此类推（注意：OpenCV 默认使用 BGR 的通道顺序，因此第一个通道通常是蓝色）。一个宽为 W、高为 H 的图像需要一个大小由 $W \times H \times 3$ 个 uchar 构成的内存块。但是，出于效率的考虑，每行会填补一些额外像素。这是因为，如果行的长度是 4 或 8 的倍数，一些多媒体处理芯片（如 Intel 的 MMX 架构）可以更高效地处理图像。这些额外的像素不会被显示或者保存，填补的值将被忽略。OpenCV 将填补后一行的长度指定为关键字。如果图像没有对行进行填补，那么图像的有效宽度就等于图像的真实宽度。成员变量 cols 代表图像的宽度（图像的列数），rows 代表图像的高度，step 代表以字节为单位的图像的有效宽度。即使你的图像的元素类型不是 uchar，step 仍然代表着行的字节数。像素的大小可以由 elemSize 函数得到：对于一个三通道的 short 型矩阵（CV_16SC3），elemSize 返回 6。图像的通道数可以由 channels 方法得到：对于灰度图像来说为 1，对于彩色图像来说为 3。total 函数返回矩阵的像素个数。

每行的像素值数可以通过如下语句得到：

```
int nc= image.cols*image.channels();
```

为了简化指针运算，cv::Mat 提供了 ptr 函数可以得到图像任意行的首地址。ptr 函数是一个模板函数，它返回第 j 行的首地址：

```
uchar* data= image.ptr<uchar>(j);
```

注意，在处理语句中，我们可以等效地使用指针运算从一列移动到下一列。所以，我们也可以这样写：

```
*data++= *data/div*div + div2;
```

扩展阅读

本例中提供的只是颜色缩减函数的一种实现方式，你不必局限于此，可以使用其他的颜色缩减公式。你还可以实现一个更通用的版本，它允许用户分别指定输入和输出图像。另外，图像遍历过程还可以通过利用图像数据的连续性，使得整个过程更高效。

最终，你甚至可以通过常规的指针运算来遍历图像。这些都将在下面讨论。

1. 其他的颜色缩减公式

在我们的例子中，颜色缩减是通过利用整数除法对结果向下取整得到的：

```
data[i] = data[i]/div*div + div/2;
```

我们也可以通过模运算来计算最接近被除数的除数（缩减因子，div）的整数倍数：

```
data[i] = data[i] - data[i]%div + div/2;
```

但是这个计算方式速度会慢一些，因为它需要存取每个像素两次。

另一个选择是使用位运算。如果我们限制缩减因子为2的幂次，即

=pow(2,n)，那么，只取像素值的前n位即可得到不大于该值的关于缩减因子的最大整数倍数。运算掩模可以通过简单的移位操作得到：

```
// 用来对像素取整的掩模  
uchar mask= 0xFF<<n;//e.g. for div=16,mask= 0xF0
```

颜色缩减可以通过如下方式计算：

```
data[i] = (data[i]&mask) + div/2;
```

通常而言，位运算非常高效，所以在需要考虑效率的情况下位运算是一个强大的备选方式。

2. 使用输入和输出参数

在我们的颜色缩减例子中，颜色变换是直接作用在输入图像上的，我们称之为In-place变换。这种方式不需要额外的图像来保存输出结果，这样可以在必要的时候节省内存。但是，在一些应用中，用户不希望原始图像被改变。这种情况下，用户不得不在调用函数之前创建一份输入图像的拷贝。注意，最简单的创建一个图像的“深拷贝”的方式是调用clone函数，如

```
// 装载图像  
image= cv::imread("boldt.jpg");  
// 克隆图像  
cv::Mat imageClone= image.clone();  
// 处理克隆图像
```

```
// 原始图像保持不变  
colorReduce(imageClone);  
// 显示处理结果  
cv::namedWindow("Image Result");  
cv::imshow("Image Result",imageClone);
```

这个额外的复制操作可以通过一种实现技巧来避免。在这种实现中，我们给用户选择到底是否采用 In-place 的处理方式。函数的实现是这样的：

```
void colorReduce(const cv::Mat &image, // 输入图像  
                 cv::Mat&result,           // 输出图像  
                 int div=64);
```

注意：输入图像是通过常量引用传递的，这意味着这个图像不会被函数修改。当选择 In-place 的处理方式时，用户可以将输入输出指定为同一个变量：

```
colorReduce(image,image);
```

否则，用户必须提供另外一个 `cv::Mat` 的实例，如

```
cv::Mat result;  
colorReduce(image,result);
```

注意：这里必须要检查输出图像和输入图像的大小和元素数据类型是否一致。方便的是，`cv::Mat` 的 `create` 成员函数内置了这个检查操作。如果需要根据新的尺寸和数据类型对一个矩阵进行重新分配，那么我们就可以调用 `create` 成员函数。而且，如果新指定的尺寸和数据类型与原有的一样，`create` 函数会直接返回，并不会对本实例做任何更改。所以，我们需要先调用 `create` 函数来创建一个与输入图像的尺寸和类型相同的矩阵：

```
result.create(image.rows,image.cols,image.type());
```

注意：`create` 函数创建的图像的内存都是连续的，`create` 函数不会对图像的行进行填补。分配的内存大小为 `total() * elemSize()`。循环使用两个指针完成：

```
for(int j=0;j<n1;j++){  
    // 得到输入输出图像的第 j 行的行首地址  
    const uchar*data_in= image.ptr<uchar>(j);
```

```
uchar* data_out= result.ptr<uchar>(j);
for(int i=0;i<nc;i++) {
    // 处理每个像素 -----
    data_out[i]=data_in[i]/div*div + div/2;
    // 像素处理完成 -----
} // 行处理结束
```

如果输入输出图像是同一幅图像，那么此函数与本例子中的第一个版本完全等价。如果二者不一样，不管事先输出图像有没有分配内存，此函数都能正常工作。

3. 高效遍历连续图像

前面我们解释过，考虑到效率，图像有可能会在行尾扩大若干个像素。但是，值得注意的是当不对行进行填补的时候，图像可以被视为一个长为 $W \times H$ 的一维数组。我们可以通过 `cv::Mat` 的一个成员函数 `isContinuous` 来判断这幅图像是否对行进行了填补。如果 `isContinuous` 方法返回值为真的话，说明这幅图像没有对行进行填补。在一些图像处理算法中，我们可以利用图像的连续性，把整个处理过程使用一个循环完成。颜色缩减函数就可以重新写为

```
void colorReduce(cv::Mat &image,int div=64) {
int nl= image.rows;// 行数
int nc= image.cols*image.channels();
if(image.isContinuous())
{
    // 没有额外的填补像素
    nc= nc*nl;
    nl= 1;    //it is now a 1D array
}
// 对于连续图像，本循环只执行 1 次
for(int j=0;j<nl;j++) {
    uchar*data= image.ptr<uchar>(j);
    for(int i=0;i<nc;i++) {
        // 处理每个像素 -----
        data[i]= data[i]/div*div+div/2;
        // 像素处理完成 -----
    } // 行处理完成
}
```

当我们通过 `isContinuous` 函数得知图像没有对行进行填补之后，我们就可以将宽设置为 1，高度设置为 $W \times H$ ，从而消除外层循环。注意，我们也可以使用 `reshape` 方法来重写这段代码：

```
if (image.isContinuous())
{
    //no padded pixels
    image.reshape(1,           // 通道数
                  image.cols*image.rows); // 行数
}
int nl= image.rows; // 列数
int nc= image.cols*image.channels();
```

`reshape` 不需要内存拷贝或者重新分配就能改变矩阵的维度。两个参数分别为新的通道数和新的行数。矩阵的列数可以根据新的通道数和行数来自适应。

在这些实现中，内层循环依次处理图像的全部像素。这个方法在同时处理若干个小图像时会很有优势。

4. 底层指针运算

在类 `cv::Mat` 中，图像数据以 `unsigned char` 形式保存在一块内存中。这块内存的首地址可以通过 `data` 成员变量得到。`data` 是一个 `unsigned char` 型的指针，所以循环可以以如下方式开始：

```
uchar *data= image.data;
```

从当前行到下一行可以通过对指针加上行宽完成：

```
data+= image.step;      // 下一行
```

`step` 代表图像的行宽（包括填补像素）。通常而言，你可以通过如下方式获得第 j 行、第 i 列像素的地址：

```
// (j,i) 处的像素地址为 &image.at(j,i)
data= image.data+j*image.step+i*image.elemSize();
```

但是，即使这种方式确实行之有效，我们依然不建议使用这种处理方式。因为这种方式除了容易出错，还不适用于带有“感兴趣区域”的图像。我们会在本章末再介绍“感兴趣区域”。

参 考

“编写高效的图像遍历循环”秘诀中对于该函数的效率有所讨论。

2.4 使用迭代器遍历图像

在面向对象的编程中，遍历数据集合通常是通过迭代器来完成的。迭代器是一种特殊的类，它专门用来遍历集合中的各个元素，同时隐藏了在给定的集合上元素迭代的具体实现方式。这种信息隐蔽原则的使用使得遍历集合更加容易。另外，不管数据类型是什么，我们都可以使用相似的方式遍历集合。标准模板库（STL）为每个容器类型都提供了迭代器。OpenCV 同样为 `cv::Mat` 提供了与 STL 迭代器兼容的迭代器。

准备工作

在这个例子中，我们依然使用前文描述的颜色缩减作为示范。

实现方法

一个 `cv::Mat` 实例的迭代器可以通过创建一个 `cv::Mat_Iterator_` 的实例来得到。类似于子类 `cv::Mat_`，下划线意味着 `cv::Mat_Iterator_` 是一个模板类。之所以如此是由于通过迭代器来存取图像的元素，就必须在编译期知道图像元素的数据类型。一个图像迭代器可以用如下方式声明：

```
cv::Mat_Iterator_<cv::Vec3b> it;
```

另外一种方式是使用定义在 `Mat_` 内部的迭代器类型：

```
cv::Mat_<cv::Vec3b>::iterator it;
```

这样就可以通过常规的 `begin` 和 `end` 这两个迭代器方法来遍历所有像素。值得指出的是，如果使用后一种方式，那么 `begin` 和 `end` 方法也必须要使用对应的模板化的版本。这样，我们的颜色缩减函数就可以重写为

```
void colorReduce(cv::Mat &image, int div=64) {  
    // 得到初始位置的迭代器
```

```
cv::Mat<cv::Vec3b>::iterator it=
    image.begin<cv::Vec3b>();
// 得到终止位置的迭代器
cv::Mat<cv::Vec3b>::iterator itend=
    image.end<cv::Vec3b>();
// 遍历所有像素
for(;it!=itend; ++it){
    // 处理每个像素 -----
    (*it)[0]= (*it)[0]/div*div+div/2;
    (*it)[1]= (*it)[1]/div*div+div/2;
    (*it)[2]= (*it)[2]/div*div+div/2;
    // 处理像素完成 -----
}
}
```

注意：因为我们这里处理的是彩色图像，所以迭代器返回的是 cv::Vec3b。每个颜色分量可以通过操作符 [] 得到。

作用原理

使用迭代器遍历任何形式的集合都遵循同样的模式。首先，创建一个迭代器特化版本的实例。在我们的示例代码中，就是 cv::Mat<cv::Vec3b>::iterator（或者 cv::Mat Iterator<cv::Vec3b>）。然后，使用集合初始位置（在我们的示例代码中，就是图像的左上角）的迭代器对其进行初始化。初始位置的迭代器通常是通过 begin 方法得到的。对于一个 cv::Mat 的实例，你可以通过 image.begin<cv::Vec3b>() 来得到图像左上角位置的迭代器。你也可以通过对迭代器进行代数运算。例如：如果你想从图像的第二行开始，那么你可以用 image.begin<cv::Vec3b>() + image.rows 来初始化迭代器。集合终止位置的迭代器可以通过 end 方法得到。但是，end 方法得到的迭代器其实已经超出了集合。这也意味着迭代过程必须在迭代器到达这个位置时结束。end 方法得到的迭代器也可以进行代数运算。例如，你希望迭代过程在图像最后一行之前停止，那么迭代器的终止位置应该是 image.end<cv::Vec3b>() - image.rows。一旦迭代器初始化完成之后，你就可以创建一个循环遍历所有的元素直至到达终止位置。一个典型的 while 循环如下所示：

```
while (it!= itend){  
    // 处理每个像素 -----  
    ...  
    // 像素处理完成 -----  
    ++it;  
}
```

操作符 `++` 用来将迭代器从当前位置移动到下一个位置。你也可以使用更大的步长，比如，用 `it+=10` 将迭代器每次移动 10px。

在循环体内部，你可以使用解引用操作符 `*` 来读写当前元素。读操作使用 `element=*it`，写操作使用 `*it=element`。注意：如果你的操作对象是 `const cv::Mat`，或者你想强调当前循环不会对 `cv::Mat` 的实例进行修改，那么你就应该创建常量迭代器。常量迭代器的声明如下：

```
cv::Mat ConstIterator_<cv::Vec3b>it;
```

或者：

```
cv::Mat_<cv::Vec3b>::const_iterator it;
```

扩展阅读

在本例中，迭代器的开始位置和终止位置是通过模板函数 `begin` 和 `end` 得到的。如同我们在本章第一则秘诀中所做的那样，我们也可以通过 `cv::Mat_` 的实例来得到它们。这样可以避免在使用 `begin` 和 `end` 方法的时候还要指定迭代器的类型。之所以可以这样，是因为一个 `cv::Mat_` 引用在创建的时候就隐式声明了迭代器的类型。

```
cv::Mat_<cv::Vec3b> cimage= image;  
cv::Mat_<cv::Vec3b>::iterator it= cimage.begin();  
cv::Mat_<cv::Vec3b>::iterator itend= cimage.end();
```

参 考

“编写高效的图像遍历循环”秘诀中对于迭代器的效率有所讨论。

如果你不熟悉面向对象编程中的迭代器的概念，你应该阅读 STL 中迭代器相关的

入门文章。用关键词“STL Iterator”进行搜索能找到很多相关的参考文献。

2.5 编写高效的图像遍历循环

在本章之前的例子中，我们介绍了几种不同的遍历图像每个像素的方法。在接下来的例子中，我们将比较这种方法的效率。当编写图像处理函数的时候，效率经常是我们关注的对象。当你设计函数的时候，需要经常检查你的代码以发现使你的程序变慢的瓶颈。

然而，很重要的一点是，除非必要，不能以降低代码清晰度的代价来换取性能的提升。简单的代码总是更容易调试和维护，只有对程序的性能造成重大影响的代码段才需要进行深度优化。

实现方法

OpenCV 有一个非常实用的函数 `cv::getTickCount()`，可以用来测量一段代码的运行时间。这个函数返回从上次开机算起的时钟周期数。由于我们需要的是某个代码段运行的毫秒数，因此还需要另外一个函数 `cv::getTickFrequency()`。此函数返回每秒内的时钟周期数。用于统计函数（或一段代码）耗费时间的方法如下：

```
double duration;
duration = static_cast<double>(cv::getTickCount());
colorReduce(image); // 被测试的函数
duration = static_cast<double>(cv::getTickCount()) - duration;
duration /= cv::getTickFrequency(); // 运行时间，以 ms 为单位
```

运行时间应该对多次运行结果求平均。在测试 `colorReduce` 这个函数时，我们又实现了一个使用 `at` 方法的版本。这个实现版本的主循环体如下：

```
for (int j=0;j<n1;j++) {
    for (int i=0;i<nc;i++) {
        //process each pixel-----
        image.at<cv::Vec3b>(j,i)[0] =
            image.at<cv::Vec3b>(j,i)[0]/div*div + div/2;
        image.at<cv::Vec3b>(j,i)[1] =
            image.at<cv::Vec3b>(j,i)[1]/div*div + div/2;
```

```

image.at<cv::Vec3b>(j,i)[2]=
    image.at<cv::Vec3b>(j,i)[2]/div*div + div/2;
//end of pixel processing-----
}//end of line
}

```

作用原理

本节中 colorReduce 函数不同实现版本的执行时间汇总如表 2.1 所示。绝对运行时间会根据不同的机器而有所不同（我们这里使用的是 Pentium 双核 2.2GHz），因此相对时间更有意义。我们测试的是在一幅分辨率为 4288×2848 的图像上运行的平均时间。测试结果总结、讨论如表 2.1 所示。

表 2.1

实现方法	平均时间
data[i]= data[i]/div*div + div/2;	37ms
*data+= *data/div*div + div/2;	37ms
*data+= v-v%div + div/2;	52ms
*data+= *data&mask + div/2;	35ms
colorReduce(input,output);	44ms
i<image.cols*image.channels();	65ms
MatIterator	67ms
.at(j,i)	80ms
3-channel loop	29ms

首先，我们对比了三种不同的颜色缩减计算方式（第 1~4 行）（详见“使用指针遍历图像”中的“扩展阅读”部分）。正如我们预见的，使用位运算的版本是最快的，耗费时间为 35ms。使用整数除法的版本消耗了 37ms，而使用模运算的版本消耗了 52ms。这意味着，最快的版本和最慢的版本之间耗费时间相差 50%！因此为图像循环体中的运算找到最高效的实现可以极大地提升程序的性能。注意，当输出图像需要被重新分配而不是以原地（In-place）方式处理时（第 5 行），运行时间为 44ms。额外的时间消耗来自内存分配。在循环体内存，对于可提前计算的变量应避免重复运算。否则会显著地增加运行时间。例如，将 colorReduce 函数内层循环中的语句：

```

int nc= image.cols*image.channels();
...
for (int i=0;i<nc;i++){

```

替换为

```
for (int i=0; i<image.cols * image.channels(); i++) {
```

这样程序会在内层循环不断地计算每行的元素个数，产生的结果是运行时间增长为 65ms，比原始版本的 35ms 要慢 80%（第 6 行）。在“使用迭代器遍历图像”秘诀中的颜色缩减函数使用了迭代器，它的速度更慢，消耗了 67ms（第 7 行）。使用迭代器的主要目标是简化图像遍历的过程从而降低出错的机会，因此没有必要对其进行优化。前一个小节中介绍的实现方法速度要慢得多（第 8 行），它消耗了 80ms。这个方法一般是用来对随机位置的像素进行读写，并不适合用来遍历图像。包含有多条语句的短循环通常而言要比只包含一条语句的长循环更为高效，即使二者处理的像素个数是相同的。同样地，如果你要对一个像素进行 N 种运算，那么在一次循环内完成要比分别用 N 个循环，每次循环只做一种运算要高效得多。为了完成同样的计算，你应该选择短循环而非长循环。例如，我们可以在内层循环内一次性处理三个通道，即一个像素，这样每行循环列数次，而不是每次只处理一个元素，却要循环列数乘以通道数次（对于彩色图像是 3）。这样，颜色缩减函数可以按如下方式重写（这是速度最快的版本）：

```
void colorReduce(cv::Mat &image, int div=64) {
    int nl= image.rows;// 行数
    int nc= image.cols;// 列数
    // 图像是连续存储的吗？
    if(image.isContinuous()) {
        // 没有对行进行填补
        nc= nc*nl;
        nl= 1;      // 一维数组
    }
    int n= static_cast<int>(
        log(static_cast<double>(div))/log(2.0));
    // 用来对像素值进行取整的二进制掩模
    uchar mask= 0xFF<<n;// e.g. for div=16,mask= 0xF0
    // for all pixels
    for(int j=0;j<nl;j++){
        // 第 j 列的地址
        uchar*data= image.ptr<uchar>(j);
        for(int i=0;i<nc;i++){
            // 处理每个像素 -----
```

```
    *data++= *data&mask + div/2;
    *data++= *data&mask + div/2;
    *data++= *data&mask + div/2;
    // 像素处理结束 -----
} // 行处理结束
}
}
```

这样的改动可以将运行时间降低到 29ms（第 9 行）。代码中也加入了图像连续存储与否的检测，可以将原本的行列双重循环精简为单层循环。对于像我们测试中使用的大尺寸图像，这项优化不是很明显，但是通常而言这个优化策略会带来很大的速度提升。

扩展阅读

多线程处理是另外一个提升算法效率的方式，尤其随着多核处理器的普及，这点变得更为重要。OpenMP 与 Intel Threading Building Blocks（TBB）是两个比较流行的并行编程 API。

参 考

参考秘诀“进行简单的图像算术计算”，它利用 OpenCV 2 的图像算术操作符来实现颜色缩减函数。

2.6 遍历图像和邻域操作

在图像处理中，通过当前位置的相邻像素计算新的像素值是很常见的操作。当邻域包含图像的前几行和下几行时，你就需要同时扫描图像的若干行。本例子向你展示如何做到这一点。

准备工作

这个例子对图像进行锐化。它基于拉普拉斯算子（将在第 6 章讨论）。众所周知，将一幅图像减去它经过拉普拉斯滤波之后的图像，这幅图像的边缘部分将得到放大，

即细节部分更加锐利。这个锐化算子的计算方式如下：

```
sharpened_pixel= 5*current-left-right-up-down;
```

其中，left 代表在当前像素左边紧挨着它的像素，up 代表在当前像素上边紧挨着它的像素，以此类推。

实现方法

这次图像处理不能以 In-place 的方式完成，用户必须提供一个输出图像。图像遍历使用到了三个指针：一个指向当前行，一个指向上一行，一个指向下一行。由于每个像素值的计算都需要它的上下左右四个邻居像素，所以不可能对图像的第一行、最后一行、第一列、最后一列进行计算。循环体的代码如下：

```
void sharpen(const cv::Mat &image, cv::Mat &result) {
    // 如有必要则分配图像
    result.create(image.size(), image.type());
    for(int j= 1;j<image.rows-1;j++) { // 处理除了第一行和最后一行之外的所有行
        const uchar* previous=
            image.ptr<const uchar>(j-1); // 上一行
        const uchar* current=
            image.ptr<const uchar>(j); // 当前行
        const uchar* next=
            image.ptr<const uchar>(j+1); // 下一行
        uchar* output= result.ptr<uchar>(j); // 输出行
        for(int i=1;i<image.cols-1;i++) {
            *output+= cv::saturate_cast<uchar>(
                5*current[i]-current[i-1]
                -current[i+1]-previous[i]-next[i]);
        }
    }
    // 将未处理的像素设置为 0
    result.row(0).setTo(cv::Scalar(0));
    result.row(result.rows-1).setTo(cv::Scalar(0));
    result.col(0).setTo(cv::Scalar(0));
    result.col(result.cols-1).setTo(cv::Scalar(0));
}
```

如果在测试图像的灰度版本上运行此函数，可以得到如图 2.4 所示的图像。



图 2.4

作用原理

为了读写当前像素上下两行的邻居像素，必须同时定义额外的指针来指向上下两行。这两个指针与当前行的指针同步增长，你才能在遍历时同时读写这三行的像素。在计算输出像素值时，模板函数 `cv::saturate_cast` 被用来对计算结果进行截断。这是因为对像素值进行数学计算经常会导致结果超出像素允许的取值范围，即小于 0 或者大于 255。解决方法是将值映射到 0~255。通常的做法是将负值截断为 0，将大于 255 的值截断为 255。这也正是函数 `cv::saturate_cast<uchar>` 所做的。另外，如果输入参数是浮点数的话，它会对其取整至最接近输入值的整数。很明显，你可以使用此函数不同类型的特化版本来将结果值限制在合理的范围之内。

边缘像素之所以不能处理是因为它们的邻域不完整，所以需要单独处理。这里我们只是简单地将其设置为 0。在其他情况下，也可以对边缘像素做特殊处理。但是，在大多数情况下，我们都没有必要花时间处理这很少的几个像素。在我们的函数实现中，我们使用了两个特殊的函数将其设置为 0。第二个是 `row` 和 `col` 方法。它们返回一个特殊的、仅包含一行（或一列）的 `cv::Mat` 实例。在这个过程中，没有任何形式的数据拷贝发生。如果这个一维矩阵的元素遭到了修改，那么原始图像的对应元素也会相

应改变。我们正是利用这个特性来修改原始矩阵的值。我们使用 `setTo` 函数来设置矩阵的值，这个函数会将矩阵的所有元素都设为指定的值。因此，这条语句：

```
result.row(0).setTo(cv::Scalar(0));
```

将结果图像的第一行的所有像素都设置为 0。对于一个三通道的彩色图像，你需要使用 `cv::Scalar(a,b,c)` 来指定像素三个通道的目标值。

扩展阅读

当计算是在像素邻域上完成时，通常可以将其用一个核矩阵表示。核描述了牵扯到的像素在计算过程中是如何组合从而获得目标值的。对于本例中的锐化滤波器，核矩阵为

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

除非特别声明，当前像素点对应核矩阵的中心。核的每一项代表对应像素的乘数因子。核在每个像素上的输出等于各个像素与对应因子乘积之和。核的尺寸等于邻域的尺寸（这里是 3×3 ）。使用这种描述形式，可以看出，锐化滤波器的输出等于 4 个水平和竖直邻居像素乘以 -1，加上当前像素乘以 5。将这么一个核应用到图像上就没有这么简单了，这是信号处理中卷积的基础。一个核定义了一个图像滤波器。由于滤波是一种常规的图像处理方法，OpenCV 定义了一个特殊的函数来完成滤波处理：`cv::filter2D`。

在使用它之前，必须先以矩阵的形式定义一个核。之后以一幅图像和这个核为参数调用此函数，函数返回滤波后的图像。利用这个函数，我们可以很简单地重写图像锐化函数：

```
void sharpen2D(const cv::Mat &image, cv::Mat &result){  
    // 构造核（所有项都初始化为 0）  
    cv::Mat kernel(3,3,CV_32F, cv::Scalar(0));  
    // 对核元素进行赋值  
    kernel.at<float>(1,1)= 5.0;  
    kernel.at<float>(0,1)= -1.0;  
    kernel.at<float>(2,1)= -1.0;
```

```
kernel.at<float>(1, 0) = -1.0;
kernel.at<float>(1, 2) = -1.0;
// 对图像进行滤波
cv::filter2D(image, result, image.depth(), kernel);
}
```

这个实现方式跟先前的实现方式的输出结果完全相同（效率也完全相同）。但是，如果核的尺寸更大，使用函数 filter2D 会更高效一些。

参 考

第 6 章对图像滤波有更详尽的介绍。

2.7 进行简单的图像算术

图像可以以不同的方式组合。因为它们只是一般的矩阵，所以它们可以做加减乘除运算。OpenCV 提供了各种图像算术操作符。我们在本例中讨论它们的用法。

准备工作

我们将第二幅图像和输入图像，通过算术操作符结合起来。第二幅图像如图 2.5 所示。

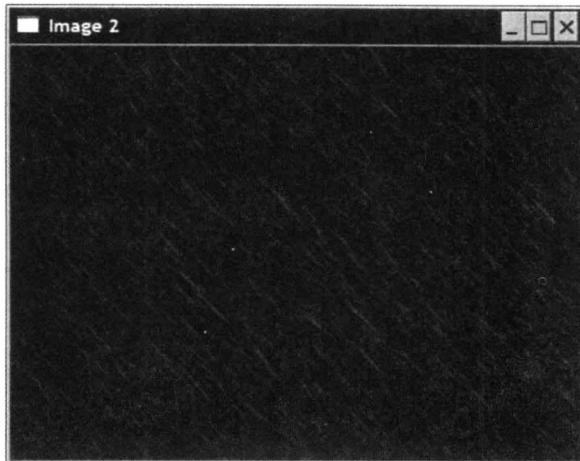


图 2.5

实现方法

首先是图像相加。当我们需要一些图像特效或者在图像上叠加信息时，就需要用到图像加法。我们通过调用函数 `cv::add`，更准确地说是函数 `cv::addWeighted` 来完成图像加法，因为我们需要的是加权和。函数调用如下：

```
cv::addWeighted(image1, 0.7, image2, 0.9, result);
```

产生的新图像屏幕截图如图 2.6 所示。



图 2.6

作用原理

所有的二元算术函数工作方式都是一样的，它接受两个输入变量和一个输出变量。在一些情况下，还需要指定权重作为运算中的标量因子。每种函数都有几个不同的形式，`cv::add` 是一个很好的例子：

```
//c[i]=a[i]+b[i];
cv::add(imageA,imageB,resultC);
//c[i]=a[i]+k;
cv::add(imageA, cv::Scalar(k), resultC);
//c[i]= k1*a[i]+k2*b[i]+k3;
cv::addWeighted(imageA, k1, imageB, k2, k3, resultC);
//c[i]= k*a[i]+b[i];
cv::scaleAdd(imageA, k, imageB, resultC);
```

对于某些函数，你可以指定一个图像掩模：

```
//if(mask[i]) c[i]= a[i]+b[i];
cv::add(imageA,imageB,resultC,mask);
```

如果指定了图像掩模，那么运算会只在掩模对应像素不为 null 的像素上进行（掩模必须是单通道的）。除了 add 之外，cv::subtract、cv::absdiff、cv::multiply 和 cv::divide 函数也有几种不同的变形。OpenCV 还提供了位运算函数：cv::bitwise_and、cv::bitwise_or、cv::bitwise_xor 和 cv::bitwise_not。cv::min 和 cv::max 也非常有用，它们用来找到矩阵中最小或者最大的像素值。所有的这些运算都使用 cv::saturate_cast（见前一个例子）来保证输出图像的像素值在合理范围以内（不会向上溢出或向下溢出）。参与运算的图像必须相同的大小和类型（输出图像如果格式不符，那么它会被重新分配）同样，由于运算是逐像素进行的，输入图像之一也可以作为输出图像。另外还有一些只接受一个输入的操作符，如 cv::sqrt、cv::pow、cv::abs、cv::cuberoot、cv::exp 和 cv::log。事实上，OpenCV 几乎拥有所有你需要的图像操作符。

扩展阅读

我们同样可以在 cv::Mat 的实例，甚至是 cv::Mat 实例的通道上使用 C++ 的算术操作符。接下来的两个小节解释了用法。

1.重载图像操作符

方便的是，大多数算术函数在 OpenCV2 中都有对应重载的操作符。因此，前面对 cv::addWeighted 的调用可以很方便地写为

```
result= 0.7*image1+0.9*image2;
```

这种简洁的写法更容易阅读。这两种写法是完全等价的。尤其值得指出的是，两种写法都会调用 cv::saturate_cast。大多数的 C++ 操作符都被重载了，如位操作符 &、|、^、~；函数 min、max、abs；比较操作符 <、<=、==、!=、>、>=。比较操作符返回一个 8 位二进制图像。同样，还有矩阵乘法 m1*m2（m1、m2 都是 cv::Mat 的实例）、矩阵求逆 m1.inv()、矩阵转置 m1.t()、矩阵的行列式 m1.determinant()、向量模 v1.norm()、向量叉乘 v1.cross(v2)、向量点积 v1.dot(v2) 等。在有意义的前提下，还有形如 op=（如 +=）的操作符。

在秘诀“编写高效的图像遍历循环”中，展示了使用循环遍历实现颜色缩减函数。根据刚刚学到的知识，利用算术操作符，这个函数可以简单地重写为

```
image=(image&cv::Scalar(mask,mask,mask))  
+cv::Scalar(div/2,div/2,div/2);
```

使用 `cv::Scalar` 是因为我们在处理彩色图像，进行“编写高效的图像遍历循环”中的性能测试之后，得到以上实现的运行时间为 89ms。速度之所以这么慢是因为这个表达式需要调用两个函数：位运算和标量和运算（而不是在一个循环体内完成所有的运算）。即使这样书写代码并非是最优的，使用图像操作符可以使代码变得简洁，程序员的生产力也更高，所以在大多数情况下你还是应该考虑使用它们。有些时候你需要分别处理一幅图像的不同通道。例如：你想只在图像的一个通道上进行一项运算。当然你可以使用图像遍历来完成这项操作，但是你也可以先使用函数 `cv::split` 将彩色图像的三个通道分别拷贝到三个独立的 `cv::Mat` 实例中，然后在对这个通道单独处理。假设我们想把前面使用到的雨滴图像只叠加到蓝色通道上，那么可以通过如下代码实现：

```
// 创建一个图像向量  
std::vector<cv::Mat> planes;  
// 将一个三通道图像分离为三个单通道图像  
cv::split(image1,planes);  
// 将新图层叠加到蓝色通道  
planes[0] += image2;  
// 将三个单通道图像重新合并为一个三通道图像  
cv::merge(planes,result);
```

函数 `cv::merge` 是 `cv::split` 的对偶运算，它将三个单通道图像合并为一个彩色三通道图像。

2.8 定义感兴趣区域

准备工作

假设我们想合并两个不同大小的图像。例如：我们想把图 2.7 所示的这个小小的 Logo 合并到我们的测试图像上。



图 2.7

由于 `cv::add` 要求两个输入图像具有相同的尺寸，所以我们不能直接使用 `cv::add`，而是需要在使用之前先定义感兴趣区域（ROI）。只要感兴趣区域的大小与 Logo 图像的大小相同，`cv::add` 就能够工作了。ROI 的位置决定了 Logo 图像被插入的位置。

实现方法

首先要定义 ROI。一旦定义之后，ROI 就可以被当做一个普通的 `cv::Mat` 实例来处理。关键之处是，ROI 和它的父图像指向同一块内存缓冲区。插入 Logo 的操作可以通过如下代码完成：

```
// 定义图像 ROI  
cv::Mat imageROI;  
imageROI= image(cv::Rect(385,270,logo.cols,logo.rows));  
// 插入 logo  
cv::addWeighted(imageROI,1.0,logo,0.3,0.,imageROI);
```

之后，得到图 2.8 所示的图像。

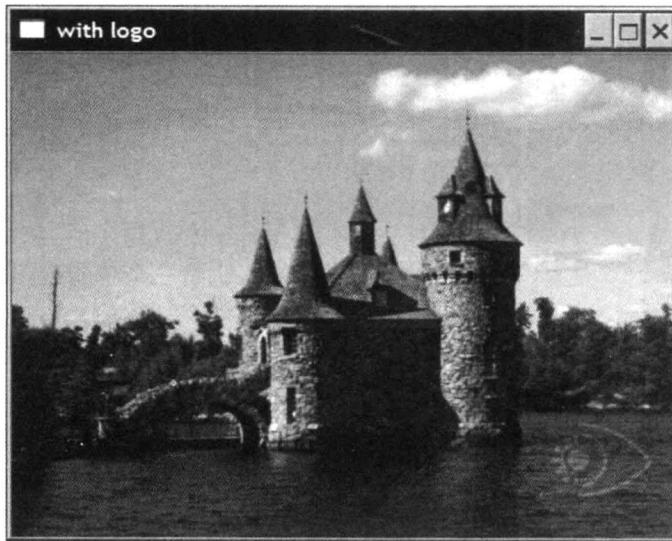


图 2.8

由于 Logo 图像是直接和原始图像相加的（同时可能会伴随着像素饱和），视觉效果不是很令人满意。所以直接将插入处的像素设置为 Logo 图像的像素值效果会好一点。

你可以通过使用一个图像掩模来完成，代码如下：

```
// 定义 ROI  
imageROI= image(cv::Rect(385,270,logo.cols,logo.rows));  
// 加载掩模（必须是灰度图）  
cv::Mat mask= cv::imread("logo.bmp",0);  
// 通过掩模拷贝 ROI  
logo.copyTo(imageROI,mask);
```

结果，图像如图 2.9 所示。

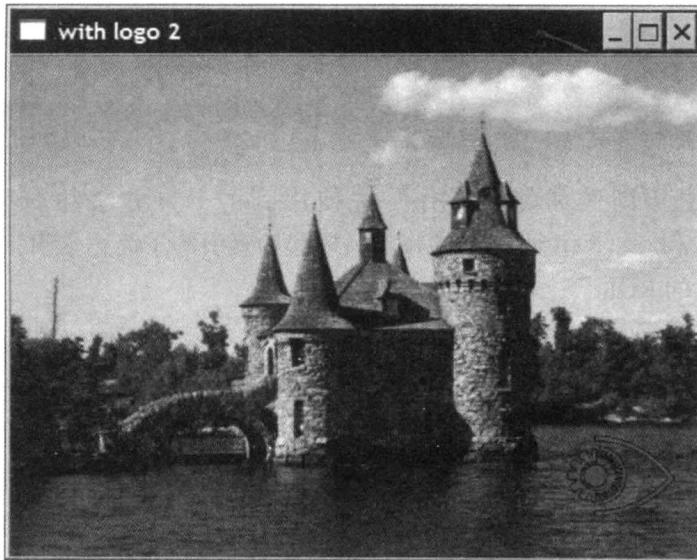


图 2.9

作用原理

定义 ROI 的一种方法是使用 `cv::Rect`。顾名思义，`cv::Rect` 表示一个矩形区域。指定矩形的左上角坐标（构造函数的前两个参数）和矩形的长宽（构造函数的后两个参数）就可以定义一个矩形区域。

另一种定义 ROI 的方式是指定感兴趣行或列的范围（Range）。Range 是指从起始索引到终止索引（不包含终止索引）的一段连续序列。`cv::Range` 可以用来定义 Range。如果使用 `cv::Range` 来定义 ROI，那么前例中定义 ROI 的代码可以重写为

```
cv::Mat imageROI= image(cv::Range(270,270+logo.rows),  
                         cv::Range(385,385+logo.cols))
```

`cv::Mat` 的 `()` 操作符返回另一个 `cv::Mat` 实例，这个实例可以用在接下来的函数调用中。因为 ROI 和原始图像共享数据缓冲区，对 ROI 的任何变换都会影响到原始图像的对应区域。由于创建 ROI 时不会拷贝数据，所以不论 ROI 的大小如何，创建 ROI 的运行时间都是常量。

如果想创建包含原始图像特定行的 ROI，可以使用如下代码：

```
cv::Mat imageROI= image.rowRange(start,end);
```

类似地，对于列：

```
cv::Mat imageROI= image.colRange(start,end);
```

在秘诀“遍历图像和邻域操作”中使用到的 `row` 方法和 `col` 方法其实是 `rowRange` 方法和 `colRange` 方法的特例，即起始索引等于终止索引，等于是定义了一个单行或单列的 ROI。

第 3 章

基于类的图像处理

本章主要探讨：

- ◆ 在算法设计中使用策略（Strategy）模式；
- ◆ 使用控制器（Controller）实现模块间通信；
- ◆ 使用单件（Singleton）设计模式；
- ◆ 使用模型 - 视图 - 控制器（Model-View-Controller）架构设计应用程序；
- ◆ 颜色空间转换。

3.1 引言

计算机视觉程序的质量与良好的编程习惯紧密相关。创建不含 Bug 的应用仅仅是开始，我们希望应用程序能够轻松地应付新的需求。本章将展示一些面向对象编程的原理，充分利用它们有助于构建高质量的代码。我们将特别介绍一些重要的设计模式，能够帮助你构建易测试、易维护、可重用的代码库。

设计模式是软件工程中众所周知的概念。一个设计模式是一个可靠的、可重用的方案，用于解决软件设计中频繁出现的问题。许多软件模式已经被详细地描述及介绍过，优质的程序员应该在工作中熟悉这些现有的模式。

本章还有一个目标，即教你如何操作图像中的颜色。贯穿于整个章节的范例将展示如何检测拥有特定颜色的像素，最后一则秘诀将解释不同的颜色空间。

3.2 在算法设计中使用策略（Strategy）模式

策略设计模式的目标是将算法封装在类中。因此，可以更容易地替换一个现有的算法，或者组合使用多个算法以拥有更复杂的处理逻辑。此外，该模式将算法的复杂度隐藏在易用的编程接口背后，降低了算法的部署难度。

准备工作

比方说，我们需要构建一个简单的算法，它可以鉴别出图像中含有给定颜色的所有像素。该算法输入的是图像以及颜色，并返回表示含有指定颜色的像素的二值图像。该算法还需要指定另外一个参数，即对颜色偏差的容忍度。

实现方法

该算法的核心部分非常简单，包含一个遍历每个像素的简单循环，将像素的颜色与目标色比较。使用上一章的知识，可以如此编写循环：

```
// 得到迭代器
cv::Mat<cv::Vec3b>::const_iterator it=
    image.begin<cv::Vec3b>();
cv::Mat<cv::Vec3b>::const_iterator itend=
    image.end<cv::Vec3b>();
cv::Mat<uchar>::iterator itout=
    result.begin<uchar>();

// 对于每个像素
for(;it!=itend;++it,++itout){
    // 处理每个像素 -----
    // 计算离目标颜色的距离
    if(getDistance(*it)<minDist){
        *itout= 255;
    }else{
        *itout= 0;
    }
    // 结束像素处理 -----
}
```

`cv::Mat` 类型的变量 `image` 表示输入图像，而 `result` 表示的是二值输出图像。因此，第一步包括初始化所需的迭代器，之后的循环遍历很容易实现。每个迭代检查当前像素的颜色与目标颜色的距离，判断是否在 `minDist` 所定义的容忍度之内。如果判断为真，那么输出图像中的当前像素赋值为 255（白色），否则赋值为 0（黑色）。`getDistance` 方法用于计算两个颜色之间的距离。计算该距离还有其他方法，如计算由 RGB 三分量的欧拉距离。本例中，为了计算简单而高效，我们仅仅是累加了 RGB 分量插值的绝对值（这也被称为街区距离）。`getDistance` 方法的定义如下：

```
// 计算与目标颜色的距离
int getDistance(const cv::Vec3b& color) const{
    return abs(color[0]-target[0])+
        abs(color[1]-target[1])+
        abs(color[2]-target[2]);
}
```

注意：我们使用 `cv::Vec3b` 来保存表示 RGB 分量的三个无符号字符。显而易见，变量 `target` 表示特定的目标色，并且正如我们将看到的，它被定义为算法类的成员变量。现在让我们完成 `process` 方法的定义。用户将提供一个输入图像，一旦对图像的遍历结束，结果也将返回：

```
cv::Mat ColorDetector::process(const cv::Mat &image) {
    // 按需重新分配二值图像
    // 与输入图像的尺寸相同，但是只有一个通道
    result.create(image.rows,image.cols,CV_8U);
```

之前的循环处理位于此：

```
...
return result;
}
```

这个方法执行时，会检查输出图像是否需要重新分配大小。这是我们使用 `cv::Mat` 的 `create` 方法的原因。记住，重新分配只在指定的尺寸和深度与当前图像结构不符时才发生。

既然我们已经定义好核心的处理方法了，让我们查看为了使用该算法还需要添加哪些额外的方法。之前我们确定了所需的输入和输出数据。因此开始定义类中保存数据的属性：

```
class ColorDetector{
private:
    // 最小可接受距离
    int minDist;
    // 目标色
    cv::Vec3b target;
    // 结果图像
    cv::Mat result;
```

为了创建算法类的一个实例（我们命名为 ColorDetector），需要定义一个构造函数。策略设计模式的目标之一是为了部署算法时尽可能简单，而最简单的构造函数显然是空的。它将创建类的一个实例，使它位于可用状态。接着，我们希望构造函数初始化所有输入参数至默认值（或是已知能带来好结果的值）。在我们的例子中，使用 100 作为距离通常是一个可接受的值。我们同时设置了一个默认的目标色，即黑色。我们需要确保总是以可预测的有效输入值作为开始：

```
// 空构造函数
ColorDetector():minDist(100) {
    // 初始化默认参数
    target[0]= target[1]= target[2]= 0;
}
```

用户在创建了实例之后，可以立即调用 process 方法，处理一张有效的图像以得到有效的结果。这是策略模式的另外一个目标，即确保算法总是以有效的参数运行。显而易见，该类的用户将使用自己的设置。通过调用合适的取值函数及设值函数即可实现。我们从色彩的容忍度参数开始：

```
// 设置色彩距离阈值。阈值必须是正的，否则设为 0
void setColorDistanceThreshold(int distance) {
    if(distance<0)
        distance=0;
    minDist= distance;
}
// 获取色彩距离阈值
int getColorDistanceThreshold() const{
    return minDist;
}
```

我们首先验证输入值的有效性。再一次强调，这是为了确保算法不会运行在无效状态。目标色的设置方法类似于

```
// 设置需检测的颜色
void setTargetColor(unsigned char red,
                     unsigned char green,
                     unsigned char blue){
    //BGR 顺序
    target[2]= red;
```

```
    target[1]= green;
    target[0]= blue;
}
// 设置需检测的颜色
void setTargetColor(cv::Vec3b color){
    target= color;
}
// 获取需检测的颜色
cv::Vec3b getTargetColor() const{
    return target;
}
```

值得注意的是，我们提供用户两种 `setTagertColor` 方法。前一种的三个参数分别是三个颜色分量，而后一种使用 `cv::Vec3b` 来保存颜色值。这样做的目的是降低算法使用的难度，用户可以挑选最合适的。

作用原理

一旦使用策略模式将算法封装在类中，可以通过创建它的实例来使用它。通常而言，实例都在程序初始化时被创建。算法参数的默认值可以被读取并显示。在拥有 GUI 的应用程序中，参数值可以通过不同的控件进行操作（文本框、滚动条等），这样用户可以方便地进行调整。但是在还未引入 GUI 之前（本章稍后将这么做），让我们先写一个简单的 `main` 函数来运行这个颜色检测算法：

```
int main()
{
    //1. 创建图像处理的对象
    ColorDetector cdetect;
    //2. 读取输入图像
    cv::Mat image= cv::imread("boldt.jpg");
if(!image.data)
    return 0;
    //3. 设置输入参数
    cdetect.setTargetColor(130,190,230); // 蓝天的颜色
    cv::namedWindow("result");
    //4. 处理并显示结果
    cv::imshow("result",cdetect.process(image));
    cv::waitKey();
    return 0;
}
```

运行上一章展示过的图片的彩色版，可以得到图 3.1 所示的输出结果。

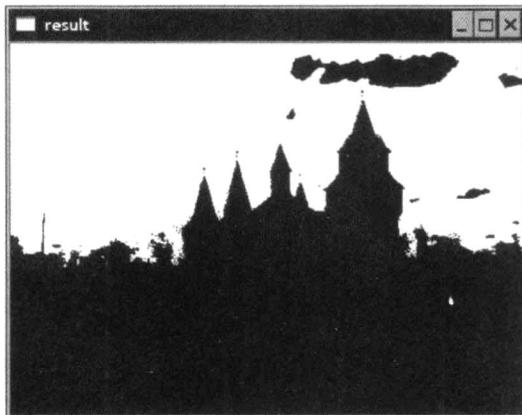


图 3.1

显而易见，这个算法相对非常简单（仅仅是一次遍历及一个容忍度参数）。当算法愈发复杂时，策略模式才能发挥出真正的威力，这样的算法涉及许多步骤，并且拥有多个参数。

扩展阅读

为了计算两个颜色向量之间的距离，我们使用下述的简单公式：

```
return abs(color[0]-target[0])+
       abs(color[1]-target[1])+
       abs(color[2]-target[2]);
```

然而，OpenCV 内置了一个函数用于计算向量的欧拉长度。因此还可以这么计算：

```
return static_cast<int>(
    cv::norm<int,3>(cv::Vec3i(color[0]-target[0],
                                color[1]-target[1],
                                color[2]-target[2])));
```

一个非常相似的结果将通过这个 `getDistance` 方法获得。这里，我们使用的是 `cv::Vec3i`（包含三个整数的向量），因为整数减法的结果还是整数。

在第 2 章中我们提到 OpenCV 的矩阵和向量结构实现一些基本的算术操作符。例如，如果你想将两个 `cv::Vec3i` 变量 `a` 和 `b` 相加，并将结果赋予 `c`，你可以简单地写：

```
c= a+b;
```

作为一种选择，计算距离的方法还能这么定义：

```
return static_cast<int>(  
    cv::norm<uchar,3>(color-target);
```

也许这个定义看上去没问题，但这是错误的。因为，所有这些操作符都包含对 `saturate_cast` 的调用（参考“遍历图像和邻域操作”）以确保结果总是在输入类型的取值范围内（这里的 `uchar`）。当 `target` 的值大于对应的 `color` 时，返回的结果将是 0，而不是误以为的负值。

参 考

A.Alexandrescu 所提出的基于策略（Policy）的类设计是策略设计模式的一个有趣变种，它的算法是在编译时期选择的。

由 Erich Gamma 等编写，Addison-Wesley 于 1994 年出版的 *Design Patterns: Elements of Reusable Object-Oriented Software* 是一本经典书籍。

推荐阅读秘诀“使用模型 - 视图 - 控制器（Model-View-Controller）架构设计应用程序”，以学习如何在 GUI 应用中使用策略模式。

3.3 使用控制器 (Controller) 实现模块间通信

当构建更复杂的应用程序时，你将需要创建多个算法，组合使用它们可以完成一些高级的任务。因此，正确地设置应用程序，在多个类之间通信，将变得越来越复杂。在单个类中集中控制整个应用程序变得很有好处，这便是控制器的目的。我们将在这个秘诀中介绍的控制器是应用程序中的一个特定对象，它扮演着重要的角色。

准备工作

创建两个按钮的简单对话框应用，其中之一用于选择图像，另一用于开始图像处理，如图 3.2 所示。

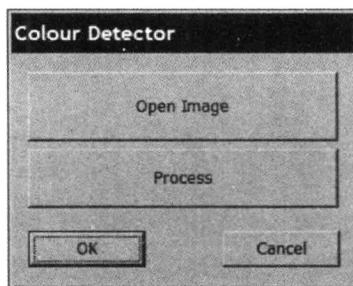


图 3.2

我们使用前一个秘诀中的 ColorDetector 类。

实现方法

控制器的角色首先是创建所需的类。这里仅有一个类。另外，我们需要两个成员变量以保存对输入及输出结果的引用。

```
class ColorDetectController{
private:
    // 算法类
    ColorDetector*cdetect;
    cv::Mat image;      // 被处理的图像
    cv::Mat result;     // 结果
public:
    ColorDetectController(){
        // 初始化工作
        cdetect=new ColorDetector();
    }
}
```

接着，你需要定义所有的设值函数和取值函数，用户将通过它们控制应用：

```
// 设置色彩距离阈值
void setColorDistanceThreshold(int distance){
    cdetect->setColorDistanceThreshold(distance);
}

// 获取色彩距离阈值
int getColorDistanceThreshold() const{
    return cdetect->getColorDistanceThreshold();
}
```

```
// 设置要检测的色彩
void setTargetColor(unsigned char red,
                     unsigned char green,unsigned char blue){
    cdetect->setTargetColor(red,green,blue);
}

// 获取需检测的颜色
void getTargetColor(unsigned char &red,
                     unsigned char &green,unsigned char &blue) const{
    cv::Vec3b color= cdetect->getTargetColor();
    red= color[2];
    green= color[1];
    blue= color[0];
}

// 设置输入图像，通过文件读取
bool setInputImage(std::string filename){
    image= cv::imread(filename);
    if (!image.data)
        return false;
    else
        return true;
}

// 返回当前的输入图像
const cv::Mat getInputImage() const{
    return image;
}
```

同时，你还需要一个调用后开始处理过程的方法：

```
// 开始图像处理
void process(){
    result= cdetect->process(image);
}
```

以及一个获取处理结果的方法：

```
// 获取最近一次处理的结果
const cv::Mat getLastResult() const{
    return result;
}
```

最后，在当应用终止时必须清理现场（同时控制器被释放）：

```
// 删除当前控制器创建的处理对象
~ColorDetectController() {
    delete cdetect;
}
```

作用原理

使用上述控制器类，一个程序员可以轻易地构建出调用算法的接口。程序员不需要理解每个类是如何联系的，或是找出哪些类中的哪些方法必须被调用以正常运作，这一切都由控制器类完成。唯一的要求是创建该控制器类的实例。

控制器中定义的设值函数和取值函数是部署算法时需要考虑的。这些方法简单地调用合适的类中的相应方法。再次强调，这个简单的例子仅包含一个类，但是在大多数情况下，将涉及多个类的实例。因此，控制器的角色是将对合适类的请求重定向，简化这些类的接口。作为这种简化的例子，考虑 `setTargetColor` 方法及 `getTargetColor`。它们都使用 `uchar` 来设置及获取感兴趣的颜色，这使得应用程序员不需要了解 `cv::Vec3b` 的任何细节。

在某些情况下，控制器还对应用程序员提供的数据进行预处理。这便是我们在 `setInputImage` 方法中所做的，对应于给定文件名的图像被加载到内存中。该方法返回 `true` 或 `false`，取决于加载操作成功与否（也可以抛出一个异常来处理这种情况）。

最后，`process` 方法负责调度整个算法。该方法不返回结果，必须调用另一个方法以获得最新一次处理的结果。

现在，创建一个非常基本的基于对话框的应用程序来使用此控制器，你只需在对话框类中添加一个 `ColorDetectController` 成员变量（这里被称为 `colordetect`）。对于 MFC 对话框而言，`Open` 按钮看起来如下：

```
//Open 按钮的回调函数
void OnOpen()
{
    // 用于选择 bmp 或 jpg 文件的 MFC 控件
    CFileDialog dlg(TRUE, _T("*.bmp"), NULL,
        OFN_FILEMUSTEXIST|OFN_PATHMUSTEXIST|OFN_HIDEREADONLY,
        _T("image files (*.bmp; *.jpg)
            (*.bmp; *.jpg|All Files (*.*)|*.*)"), NULL);
    dlg.m_ofn.lpstrTitle= _T("Open Image");
```

```
// 如果选中了一个文件
if(dlg.DoModal() == IDOK) {
    // 获取选中文件的路径
    std::string filename= dlg.GetPathName();
    // 设置并显示该图像
    colordetect.setInputImage(filename);
    cv::imshow("Input Image", colordetect.getInputImage());
}
}
```

第二个按钮执行处理过程，并显示结果：

```
//Process 按钮的回调函数
void OnProcess()
{
    // 此处目标色为硬编码
    colordetect.setTargetColor(130,190,230);
    // 处理输入图像，并显示结果
    colordetect.process();
    cv::imshow("Output Result", colordetect.getLastResult());
}
```

显然，一个更完整的应用程序将包括额外的控件以允许用户设置算法参数。

参 考

“使用模型 - 视图 - 控制器（Model-View-Controller）架构设计应用程序”展示了使用 GUI 的更扩展的范例。

3.4 使用单件 (Singleton) 设计模式

单件是另外一种流行的设计模式，用于简化对一个类实例的访问，同时保证在程序执行期间只有一个实例存在。在这则秘诀中，我们使用单件来访问一个控制器对象。

准备工作

我们使用上一则秘诀中的 ColorDetectController 类。该类将被修改，以包含一个单件类。

实现方法

要做的第一件事情是添加一个私有的静态成员变量，它将保存对单个类实例的引用。同时，为了禁止创建额外的类实例，构造函数也是私有的：

```
class ColorDetectController {  
private:  
    // 单件指针  
    static ColorDetectController *singleton;  
    ColorDetector *cdetect;  
    // 私有构造函数  
    ColorDetectController() {  
        // 初始化工作  
        cdetect= new ColorDetector();  
    }  
}
```

此外，你还可以使复制构造函数和操作符 = 私有化，以确保无法创建独一无二的单件实例的拷贝。当一个用户的类要求单件类的一个实例时，它才被创建。这通过使用一个静态方法实现，如果实例不存在那么创建它，然后返回一个指向该实例的指针：

```
// 访问单件实例  
static ColorDetectController *getInstance(){  
    // 在首次调用时创建实例  
    if(singleton == 0)  
        singleton= new ColorDetectController;  
    return singleton;  
}
```

需要注意的是，单件的实现并不是线程安全的。因此，在多线程情况下不应该使用它。

最后，因为单件实例是被动态创建的，当不需要时用户必须删除它。这也是通过一个静态方法实现的：

```
// 释放单件的实例  
static void destroy(){  
    if(singleton != 0){  
        delete singleton;  
    }  
}
```

```
    singleton= 0;  
}  
}
```

由于单件是一个静态成员变量，它必须在 .cpp 文件中定义，如下：

```
#include "colorDetectController.h"  
ColorDetectController *ColorDetectController::singleton=0;
```

作用原理

因为单件可以通过一个公共的静态方法获取，所有包括单件类声明的类都能访问它。这尤其适用于控制器对象，它被多个拥有复杂 GUI 的窗口控件类访问。其中的任何一个 GUI 类都不需要声明一个成员变量，这和前一则秘诀不同。对话框类的两个回调方法编写如下：

```
//Open 按钮的回调函数  
void OnOpen()  
{  
    ...  
    // 如果选中了一个文件  
    if(dlg.DoModal() == IDOK){  
        // 获取选中文件的路径  
        std::string filename= dlg.GetPathName();  
        // 设置并显示该图像  
        ColorDetectController::getInstance()->setInputImage(filename);  
        cv::imshow("Input Image",  
                  ColorDetectController::getInstance()->getInputImage());  
    }  
}  
//Process 按钮的回调函数  
OnProcess()  
{  
    // 此处目标色为硬编码  
    ColorDetectController::getInstance()->setTargetColor(130,190,230);  
    // 处理输入图像，并显示结果
```

```
ColorDetectController::getInstance()->process();
cv::imshow("Output Result",
ColorDetectController::getInstance()->getLastResult());
}
```

当应用程序关闭时，单件的实例必须被释放：

```
//Close 按钮的回调函数
void OnClose()
{
    // 释放单件
    ColorDetectController::getInstance()->destroy();
    OnOK();
}
```

如上所示，当一个控制器被封装在一个单件中，它变得更容易访问。然而，一个更好的实现需要一个更复杂的 GUI。这将在下一则秘诀中实现，通过展示模型 - 视图 - 控制器架构，概括了在应用程序设计中使用设计模式的讨论。

3.5 使用模型-视图-控制器 (Model-View-Controller) 架构设计应用程序

之前的秘诀让你认识了三个重要的设计模式：策略、控制器及单件。这则秘诀将介绍的架构模式，将联合使用这三个模式以及其他类。这便是模型 - 视图 - 控制器 (MVC)，它的目的是够清晰地分离程序中的逻辑部分与用户交互部分。在这则秘诀中，我们将使用 MVC 模式来构建一个基于 Qt 的图形界面应用程序。然而，在开始编程之前，让我们给这个模式一个简短的描述。

准备工作

顾名思义，MVC 模式包括三个主要组件。首先来看一看每个组件的角色。

模型 (Model) 包含关于应用程序的信息，它拥有所有由应用程序处理的数据。当出现新的数据，它将告知控制器，后者会要求 View 来显示新的结果。通常，模型将集合多个算法，它们很有可能按照策略模式进行实现。所有这些算法都是模型的一部分。

视图(View)对应于用户界面。它是由不同的控件组成的，这些控件向用户显示数据，并允许用户与应用程序交互。它的职责之一是发送用户的命令到控制器。当新数据可用时，它也会刷新自己以显示新的信息。

控制器 (Controller) 将视图和模型桥接在一起。它接收视图的请求，将其转化为模型中的合适方法。它也会在模型更改状态时得到通知，并因此请求视图刷新以显示新的信息。

实现方法

就像我们在上一节所做的，我们将使用 ColorDetector 类。这将是我们自己的模型，它包含应用程序的逻辑和底层数据。我们还实现了一个控制器，即 ColorDetectController 类。然后选择最合适的控件，使创建一个更复杂的 GUI 变得简单。例如，使用 Qt 可以建立图 3.3 所示的界面。

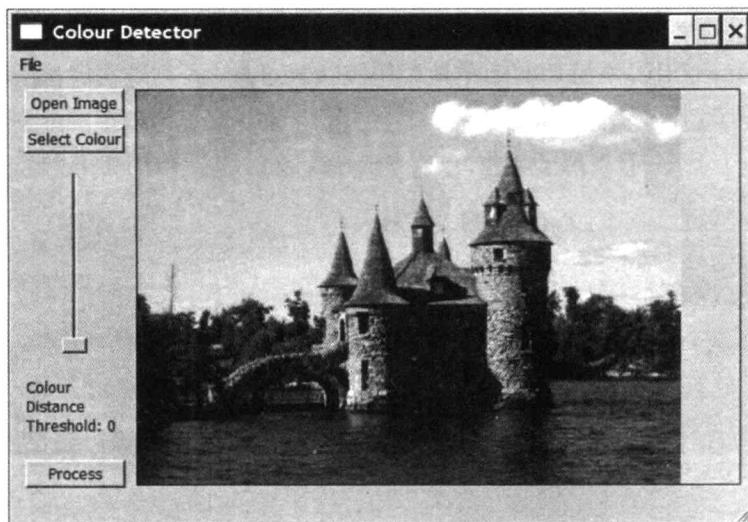


图 3.3

Open Image 按钮用于选择并打开一个图像。待检测的颜色可以通过按 Select Color 按钮进行选择，它将打开一个颜色选取控件（在下方以黑白两色印刷）方便用户选择颜色。

滚动条被用于选择合适的阈值。然后，通过按 Process 按钮，图像被处理而结果得到显示（图 3.4）。

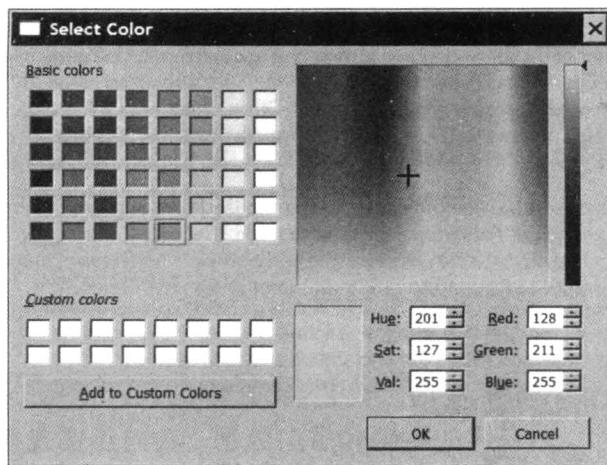


图 3.4

作用原理

MVC 架构下，用户界面只是简单地调用控制器的方法，它既不包含任何应用数据，也没有实现任何应用逻辑。因此，很容易替换接口。在这里，添加了一个颜色选取控件，即 QColorDialog，一旦选取了颜色，便在 Select Color 的选项卡中调用适当的控制器方法：

```
QColor color = QColorDialog::getColor(Qt::green, this);
if(color.isValid()){
    ColorDetectController::getInstance()
        ->setTargetColor(color.red(), color.green(), color.blue());
}
```

QSlider 控件设置阈值。当 Process 被按下时，该值被读取，这也触发了图像处理和结果显示：

```
ColorDetectController::getInstance()
    ->setColorDistanceThreshold(
        ui->verticalSlider_Threshold->value());
ColorDetectController::getInstance()->process();
cv::Mat resulting=
    ColorDetectController::getInstance()->getLastResult();
if(!resulting.empty())
    displayMat(resulting);
```

事实上，Qt 的 GUI 库大量使用 MVC 模式。它使用信号（Signal）的概念以保持所有的 GUI 控件与数据模型同步。

参 考

Qt 的在线文档能帮助你了解 MVC 模式在 Qt 中的实现 (<http://doc.qt.nokia.com>)。

阅读第 1 章的“使用 Qt 创建 GUI 应用”以初步了解 Qt GUI 框架，以及它的信号与槽模型。

3.6 颜色空间转换

本节教你如何封装算法到一个类中。通过简化后的接口，算法变得容易使用。封装还允许你修改一个算法的实现而不影响使用它的类。这个法则将在本秘诀中得到说明，我们将修改 ColorDetector 类算法以使用另一个颜色空间。因此，本节还将介绍 OpenCV 中的颜色转换。

准备工作

RGB 颜色空间（或 BGR 取决于颜色的存储顺序）基于对红、绿、蓝三原色的使用。这些颜色被选中是因为当它们结合在一起时，可以生成广阔色域中的颜色。事实上，人类的视觉系统也建立在对三原色的感知上，人眼圆锥细胞对红、绿、蓝光谱具备敏感性。RGB 空间通常是在数字图像领域的默认颜色空间，因为这正是它们被获取的方式。捕获的光线穿过红、绿、蓝滤光器。此外，在数字图像中，红色、绿色和蓝色通道经过了调整，当它们的量相等时，能够得到一个灰度强度，即从黑色 (0, 0, 0) 到白色 (255, 255, 255)。

不幸的是，使用 RGB 颜色空间来计算颜色之间的距离并不是衡量颜色相似度最好的方法。事实上，RGB 并不是一个感知上均匀分布的色彩空间。这意味着在相同的距离下，两个颜色可能看起来非常相似，而另外两个颜色却截然不同。

为了解决这个问题，人们提出了在感知上均匀分布的色彩空间。特别地，CIE L*a*b* 便是这样一个颜色空间。通过转换我们的图像到这个空间，图像像素和目标色之间的欧拉距离在描述颜色的相似性上才有意义。在这个秘诀中，我们将修改之前的代码以工作于 CIE L*a*b* 空间。

实现方法

通过使用 OpenCV 函数 `cv::cvtColor` 可以轻易在不同颜色空间之间进行转换。让我们在 `process` 方法的开始转换输入图像到 CIE L*a*b* 空间：

```
cv::Mat ColorDetector::process(const cv::Mat &image) {
    // 按需重新分配二值图像
    // 与输入图像的尺寸相同，但是只有一个通道
    result.create(image.rows, image.cols, CV_8U);
    // 按需重新分配中间图像
    converted.create(image.rows, image.cols, image.type());
    // 转换到 Lab 颜色空间
    cv::cvtColor(image, converted, CV_BGR2Lab);
    // 得到转换后图像的迭代器
    cv::Mat<cv::Vec3b>::iterator it=
        converted.begin<cv::Vec3b>();
    cv::Mat<cv::Vec3b>::iterator itend=
        converted.end<cv::Vec3b>();
    // 得到输出图像的迭代器
    cv::Mat<uchar>::iterator itout= result.begin<uchar>();
    // 对于每个像素
    for(;it!=itend;++it,++itout){
        ...
    }
}
```

变量 `converted` 包含颜色空间转换后的图像。在 `ColorDetector` 类中，它被定义为一个类属性：

```
class ColorDetector{
private:
    // 包含转换后的图像
    cv::Mat converted;
```

我们还需要转换目标色。为此，创建一个临时的只包含 1px 的图像。注意，你需要保持与之前秘诀相同的函数签名，即用户继续提供位于 RGB 空间的目标色：

```
// 设置要检测的色彩
void setTargetColor(unsigned char red,
                    unsigned char green, unsigned char blue){
```

```
// 临时的 1px 图像
cv::Mat tmp(1,1,CV_8UC3);
tmp.at<cv::Vec3b>(0,0)[0]= blue;
tmp.at<cv::Vec3b>(0,0)[1]= green;
tmp.at<cv::Vec3b>(0,0)[2]= red;
// 转换目标色到 Lab 颜色空间
cv::cvtColor(tmp,target,CV_BGR2Lab);
target= tmp.at<cv::Vec3b>(0,0);
}
```

如果之前的应用程序使用这个修改后的类进行编译，它现在将使用 CIE L*a*b* 颜色空间检测与目标色相似的像素。

作用原理

当图像从一个颜色空间转换到另一个，线性或非线性变换将作用于每个输入像素，以产生输出像素。输出图像的像素类型将匹配输入图像的像素类型之一。即使大多数时间你使用的是 8 位像素，你仍可以对浮点图像（像素值通常为位于 0~1.0 的小数）或整数图像（像素一般位于 0~65 535）使用颜色转换，但像素的确切值域取决于特定的颜色空间。例如，在 CIE L*a*b* 颜色空间，L 通道位于 0~100，而色度分量 a 和 b 则位于 -127~127。

OpenCV 提供了最常用的颜色空间。这只是一个传递正确的掩码给 OpenCV 函数的问题。其中之一是 YCrCb，该颜色空间用于 JPEG 压缩。从 BGR 到 YCrCb 转换所需的参数 CV_BGR2YCrCb。注意，三原色红、绿、蓝可以在 RGB 次序或 BGR 次序中使用。

HSV 和 HLS 颜色空间也很有用，因为它们可以分解成色调(Hue)、饱和度(Saturation)、明度 (Value) 或亮度 (Luminance) 分量。对人类而言，用它来描述颜色更加自然。

你还可以将彩色图像灰度化，输出结果将会是一个单通道的图像：

```
cv::cvtColor(color,gray,CV_BGR2Gray);
```

你也可以进行反方向的转换，但是生成的三通道彩色图像中的每个通道都同样充满着相应灰度图像中的值。

参 考

第4章中的“使用均值漂移（Mean Shift）算法查找物体”，使用HSV颜色空间以寻找图像中的物体。

关于颜色空间理论有许多优秀的参考读物，“*The Structure and Properties of Color Spaces and the Representation of Color Images*”（E.Dubois,Morgan,Claypool,Oct.2009）是一本很好的最新读物。

第4章

使用直方图统计像素

本章主要探讨：

- ◆ 计算图像的直方图；
- ◆ 通过查找表修改图像外观；
- ◆ 直方图均衡化；
- ◆ 反投影直方图以检测特定的图像内容；
- ◆ 使用均值漂移（Mean Shift）算法查找物体；
- ◆ 通过比较直方图检索相似图片。

4.1 引言

一个图像是由不同颜色值的像素组成的。像素值在图像中的分布情况是这幅图像的一个重要特征。本章介绍图像直方图的概念。你将学习如何计算并使用直方图来修改图像的外观。直方图也可用于描述图像的内容，并且检测图像中特定的对象或纹理。本章将一一展示这些技术。

4.2 计算图像的直方图

图像是由像素组成的，在一个单通道的灰度图像中，每个像素的值介于0（黑色）~255（白色）之间。根据图像的内容，你会发现每个灰度值的像素数目是不同的。

直方图是一个简单的表，它给出了一幅图像或一组图像中拥有给定数值的像素数量。因此，灰度图像的直方图有256个条目（或称为容器）。0号容器给出值为0的像素个数，1号容器给出值为1的像素个数，依此类推。显然，如果你对直方图的所有项求和，会得到像素的总数。直方图也可以被归一化，归一化后的所有项之和等于1。在这种情况下，每一项给出的都是拥有特定数值的像素在图像中占的比例。

创建一个简单的控制台项目，准备一张类似图 4.1 的图片。

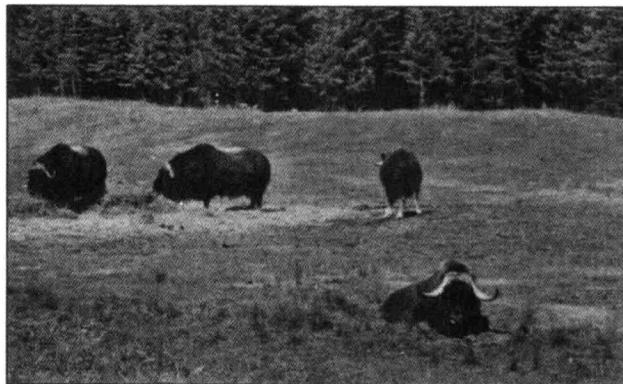


图 4.1

实现方法

在 OpenCV 中计算直方图可以通过使用 `cv::calcHist` 函数。这是一个通用函数，可计算任意像素类型的多通道图像。为了使用更简单，我们定义一个专门的类来处理单通道的灰度图像：

```
class Histogram1D{
private:
    int histSize[1]; // 项的数量
    float hranges[2]; // 像素的最小及最大值
    const float*ranges[1];
    int channels[1]; // 仅用到一个通道
public:
    Histogram1D(){
        //准备 1D 直方图的参数
        histSize[0]= 256;
        hranges[0]= 0.0;
        hranges[1]= 255.0;
        ranges[0]= hranges;
        channels[0] = 0; // 默认情况，我们考察 0 号通道
    }
}
```

定义好成员变量后，计算一个灰度直方图可通过以下方法：

```
// 计算 1D 直方图
cv::MatND getHistogram(const cv::Mat &image) {
    cv::MatND hist;
    // 计算直方图
    cv::calcHist(&image,
        1,           // 计算单张图像的直方图
        channels,    // 通道数量
        cv::Mat(), // 不使用图像作为掩码
        hist,        // 返回的直方图
        1,           // 这是 1D 的直方图
        histSize,    // 项的数量
        ranges       // 像素值的范围
    );
    return hist;
}
```

现在，你的程序只需要打开图像，创建一个 Histogram1D 实例，并调用 getHistogram 方法：

```
// 读取输入图像
cv::Mat image= cv::imread("../group.jpg",
                           0); // 以黑白模式打开
//histogram 对象
Histogram1D h;
// 计算直方图
cv::MatND histo=h.getHistogram(image);
```

这里的 histo 对象是一个拥有 256 个条目的一维数组。因此，通过遍历数组即可读取每个条目的值：

```
// 遍历每个条目
for(int i=0;i<256;i++)
    cout<<"Value"<<i<<"="<<
        histo.at<float>(i)<<endl;
```

以图 4.1 为例，显示的数值将是

```
...
Value 7= 159
Value 8= 208
```

```
Value 9= 271
Value 10= 288
Value 11= 340
Value 12= 418
Value 13= 432
Value 14= 472
Value 15= 525
...
```

很难从这组数值中提取任何直观的含义。因此，更方便的方式是将直方图可视化，如使用柱状图。下面便是一例：

```
// 计算 1D 直方图，并返回一幅图象
cv::Mat getHistogramImage(const cv::Mat &image) {
    // 首先计算直方图
    cv::MatND hist= getHistogram(image);
    // 获取最大值和最小值
    double maxVal=0;
    double minVal=0;
    cv::minMaxLoc(hist,&minVal,&maxVal,0,0);
    // 显示直方图的图像
    cv::Mat histImg(histSize[0],histSize[0],
                    CV_8U,cv::Scalar(255));
    // 设置最高点为 nbins 的 90%
    int hpt = static_cast<int>(0.9*histSize[0]);
    // 每个条目都绘制一条垂直线
    for(int h = 0;h<histSize[0];h++){
        float binVal = hist.at<float>(h);
        int intensity = static_cast<int>(binVal*hpt/maxVal);
        // 两点之间绘制一条线
        cv::line(histImg,cv::Point(h,histSize[0]),
                 cv::Point(h,histSize[0]-intensity),
                 cv::Scalar::all(0));
    }
    return histImg;
}
```

使用这种方法，你可以获得以柱状图方式显示的直方图：

```
// 以图形方式显示直方图  
cv::namedWindow("Histogram");  
cv::imshow("Histogram",  
          h.getHistogramImage(image));
```

结果如图 4.2 所示。

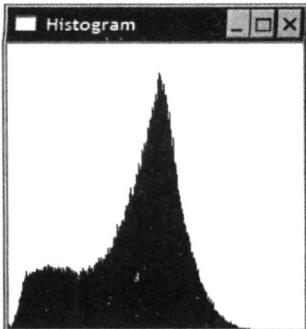


图 4.2

从这个直方图可以直观地看到，灰度的中间位置存在一个大的峰值，同时有大量深色的像素。这两组像素基本对应的是图像的背景和前景。通过在这两组像素之间的过渡处进行阈值化可以证实这一点。有一个方便的 OpenCV 函数可用于此，即 `cv::threshold` 函数。当需要使用一个阈值来创建二值图像时可以使用该函数。此处我们选择的阈值的是峰值上升前的最小值（灰度值为 60）：

```
cv::Mat thresholded;  
cv::threshold(image, thresholded, 60, 255, cv::THRESH_BINARY);
```

生成的二值图像清晰地显示了分段的背景与前景，如图 4.3 所示。



图 4.3

作用原理

`cv::calcHist` 有许多参数，因此可以使用在许多上下文中。大多数时候，你的直方图将是一个单通道或三通道的图像。然而，该函数允许你指定一个分布在几个图像中的多通道图像。这就是为何一组图像被作为该函数的输入。第 6 个参数指定直方图的维度，例如 1 指的是一维直方图。直方图计算中需要考虑的通道列在一个数组中，它有指定的维度。在实现这个单通道的类时，默认使用的是通道 0（第 3 个参数）。直方图本身是通过每个维度的条目数量、每个维度的最小及最大值进行描述的，前者位于第 7 个参数，是一组整数，后者位于第 8 个参数，是一组每项包含两个元素的数组。也可以定义一个非均匀的直方图，在这种情况下你需要指定每个条目的最大值。

和许多 OpenCV 函数一样，可以指定一个掩码，指明哪些项目需要进行统计（掩码值为 0 的像素都将被忽略）。还可以指定两个额外的可选参数，都是布尔值。第 1 个值表明直方图是否是归一化的（默认为 `true`），第 2 个值允许你积累多个直方图计算的结果。如果后一个参数是 `true`，那么图像的像素统计值会加到输入直方图中当前的值上。当有人想对一组图片计算直方图时这很有用。

返回的直方图存储在一个 `cv::MatND` 实例中。这是一个通用类，可以操作 N 维矩阵。当然，它已经定义了一维、二维、三维矩阵的 `at` 方法。因此在 `getHistogramImage` 方法访问一维直方图的每个条目时我们可以写：

```
float binVal = hist.at<float>(h);
```

需要注意的是，直方图中的值是浮点数。

扩展阅读

本节介绍的 `Histogram1D` 类通过限制仅使用一维直方图简化了 `cv::calcHist` 函数，这对于灰度图是够用的。相似的，我们可以定义一个类来计算彩色的 BGR 图像的直方图：

```
class ColorHistogram{  
private:  
    int histSize[3];  
    float hranges[2];  
    const float*ranges[3];  
    int channels[3];
```

```
public:  
    ColorHistogram() {  
        // 准备彩色直方图的参数  
        histSize[0] = histSize[1] = histSize[2] = 256;  
        hranges[0] = 0.0;           // BGR 的范围  
        hranges[1] = 255.0;  
        ranges[0] = hranges; // 所有通道拥有相同的范围  
        ranges[1] = hranges;  
        ranges[2] = hranges;  
        channels[0] = 0;          // 三个通道  
        channels[1] = 1;  
        channels[2] = 2;  
    }  
}
```

在这个例子中，直方图是三维的。因此，我们需要为每个维度指定范围。对于 BGR 图像，这三个通道有相同的 [0, 255] 范围。参数准备好后，彩色直方图通过以下方法计算：

```
cv::MatND getHistogram(const cv::Mat &image) {  
    cv::MatND hist;  
    // 计算直方图  
    cv::calcHist(&image,  
        1,                  // 仅计算一张图  
        channels,           // 通道数量  
        cv::Mat(),          // 不使用掩码图像  
        hist,               // 返回的直方图  
        3,                  // 这是三维直方图  
        histSize,            // 项的数量  
        ranges              // 像素值的范围  
    );  
    return hist;  
}
```

返回的是一个三维的 `cv::Mat` 实例。这个矩阵有 $(256)^3$ 的元素，能够表示超过 1600 万个条目。在许多应用程序中，在计算这样一个大型直方图前最好能够减少颜色的数量（参见第 2 章）。或者，你也可以使用 `cv::SparseMat` 数据结构，它旨在表示大型的稀疏矩阵（即矩阵拥有少量非零元素），同时不会消耗过多的内存。某个版本的 `cv::calcHist` 函数返回这样一个矩阵。因此，简单地修改前面的方法即可使

用 cv::SparseMatrix:

```
cv::SparseMat getSparseHistogram(const cv::Mat &image) {
    cv::SparseMat hist(3,histSize,CV_32F);
    // 计算直方图
    cv::calcHist(&image,
        1,           // 仅计算一张图
        channels,     // 通道数量
        cv::Mat(),    // 不使用图像作为掩码
        hist,         // 返回的直方图
        3,           // 这是三维直方图
        histSize,     // 项的数量
        ranges       // 像素值的范围
    );
    return hist;
}
```

参 考

本节之后将介绍的秘诀“反投影直方图以检测特定的图像内容”将基于颜色直方图进行特定图像内容的检索。

4.3 使用查找表修改图像外观

图像直方图捕捉方式呈现一个场景使用可用的像素强度值。通过分析像素值的分布在一个图像，可以使用此信息来修改，甚至可以提高一个图像。这个食谱解释了一个可以使用一个简单的映射函数，代表一个查找表，修改图像的像素值。

实现方法

查找表是一个简单的一对一（或多对一的）函数，定义了如何将像素值转换为新的值。它本质上一个一维数组，对于常规灰度图像而言有 256 个条目。表的第 i 项表示相对应灰度的新值，即：

```
newIntensity= lookup[oldIntensity];
```

OpenCV 的 `cv::LUT` 对图像应用查找表以生成新图像。我们可以添加这个功能到 `Histogram1D` 类:

```
cv::Mat applyLookUp(const cv::Mat& image, // 输入图像
                     const cv::Mat& lookup) { // 1x256 uchar matrix
    // 输出图像
    cv::Mat result;
    // 应用查找表
    cv::LUT(image, lookup, result);
    return result;
}
```

作用原理

当一个查找表应用于一个图像时，结果是一个新的图像，它的像素强度值已经按照规定的查找表进行修改。像这样的一个简单转换可以如下：

```
// 创建图像的反向查找表
int dim(256);
cv::Mat lut(1,           // 1D
            &dim,          // 256 项
            CV_8U);        // uchar
for(int i=0;i<256;i++) {
    lut.at<uchar>(i)= 255-i;
}
```

这个转换仅仅反转像素的强度，即强度 0 变为 255，1 变为 254，等等。应用这样一个查找表会在一个图像上产生原图的负片。对于之前秘诀中的图像，其结果如图 4.4 所示。



图 4.4

扩展阅读

你还可以定义一个提高图像的对比度的查找表。例如，如果你观察之前显示的图像在第一则秘诀中的原始直方图，很容易注意到强度值的部分范围并没有用到（对于这个图像，尤其是较亮部分的强度没有用到）。因此通过拉伸直方图，能得到扩展后的对比度。这个过程旨在检测直方图中非零项的最低 (`imin`) 和最高 (`imax`) 强度值。强度值可以被重新映射，这样 `imin` 值是重新定位在强度 0，`imax` 是分配值 255。简单的中间强度线性重新映射，如下：

```
255.0*(i-imin)/(imax-imin)+0.5;
```

因此，完整的图像拉伸方法定义如下：

```
cv::Mat stretch(const cv::Mat &image, int minValue=0) {
    // 首先计算直方图
    cv::MatND hist=getHistogram(image);
    // 寻找直方图的左端
    int imin= 0;
    for(;imin<histSize[0];imin++){
        std::cout<<hist.at<float>(imin)<<std::endl;
        if(hist.at<float>(imin)>minValue)
            break;
    }
    // 寻找直方图的右端
    int imax= histSize[0]-1;
    for(;imax>= 0;imax--){
        if(hist.at<float>(imax)>minValue)
            break;
    }
    // 创建查找表
    int dim(256);
    cv::Mat lookup(1, //1 dimension
                  &dim,           //256 entries
                  CV_8U);         //uchar
    // 填充查找表
    for(int i=0;i<256;i++){
        // 确保数值位于 imin 与 imax 之间
        if(i<imin) lookup.at<uchar>(i)= 0;
```

```

    else if(i>imax) lookup.at<uchar>(i)= 255;
    // 线性映射
    else lookup.at<uchar>(i)= static_cast<uchar>(
        255.0*(i-imin)/(imax-imin)+0.5);
}
// 应用查找表
cv::Mat result;
result= applyLookUp(image,lookup);
return result;
}

```

计算好查找表后我们调用 `applyLookUp` 方法。在实践中我们发现，忽略不含任何数值的条目大有益处，同时也可以忽略微不足道的条目，如小于一个给定的值（这里定义为 `minValue`）。该方法调用如下：

```

// 忽略两端不足 100 像素的项
cv::Mat stretechd= h.stretch(image,100);

```

生成的图像显示如图 4.5 所示。

扩展后的直方图如图 4.6 所示。

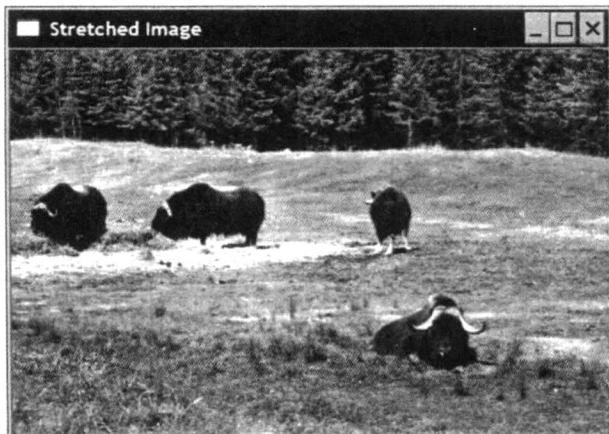


图 4.5

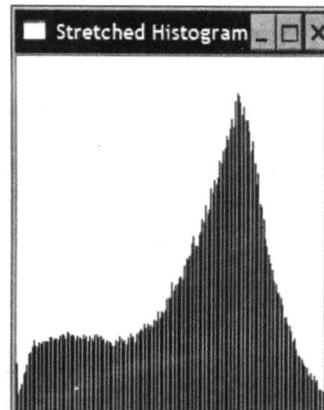


图 4.6

参 考

秘诀“直方图均衡化”展示了另一种提高图像对比度的方式。

4.4 直方图均衡化

在前面的秘诀中，我们展示了一种提高图像对比度的方法，即通过拉伸直方图，使它覆盖所有的取值范围。这个策略确实可以简单有效地提升图像质量。然而，在多数情况下，图像在视觉的缺陷并非源于使用过窄的强度范围，而是由于某些颜色值的出现频率高于另一些。本章的第一则秘诀中的直方图便是一个很好的例子，中灰色的强度在图中被大量使用，而较亮和较暗的像素值相当罕见。事实上，我们可以认为一幅高质量的图像应该平均使用所有的像素强度。这便是直方图均衡化背后的理念，即使得图像的直方图尽可能平坦。

实现方法

OpenCV 提供了一个简单易用的函数来执行直方图均衡化。它的调用方式如下：

```
cv::Mat equalize(const cv::Mat &image) {
    cv::Mat result;
    cv::equalizeHist(image, result);
    return result;
}
```

应用于我们的图像后，得到的结果如图 4.7 所示。该图的直方图如图 4.8 所示。

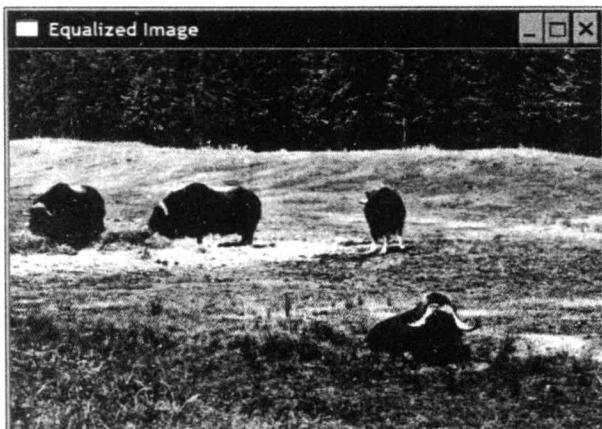


图 4.7

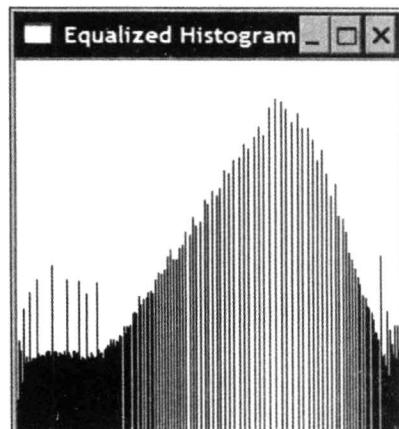


图 4.8

当然，直方图无法变成完全平坦的，因为查找表是一个全局的多对一变换。但是可以看到，现在的直方图分布比原先更均衡。

作用原理

在一个完全均衡的直方图中，所有的容器有同等数量的像素。这意味着 50% 的像素的强度低于 128，25% 的像素强度低于 64，等等。这个观察可以通过以下规则进行表达：在一个均衡的直方图中， $p\%$ 的像素的强度值必须低于或等于 $255 \times p\%$ 。这条规则用于均衡化一个直方图：强度 i 的映射值应该对应于强度值小于 i 的像素所占的比例。因此，所需的查找表可由以下等式构建：

```
lookup.at<uchar>(i) = static_cast<uchar>(255.0*p[i]);
```

此处， $p[i]$ 是强度值小于或等于 i 的像素的个数。我们通常称函数 $p[i]$ 为累积直方图，也就是说，这个直方图包含的是强度低于或等于给定数值的像素个数，而不是拥有给定数值的像素个数。

一般而言，直方图均衡化能够大幅改善图像的外观。然而，最终结果的质量依赖于图像本身的内容，有好有坏。

4.5 反投影直方图以检测特定的图像内容

直方图是图像内容的一个重要特性。如果一幅图像的区域中显示的是一种独特的纹理或是一个独特的物体，那么这个区域的直方图可以看作是一个概率函数，它给出的是某个像素属于该纹理或物体的概率。在这则秘诀中，我们将了解到如何借助图像的直方图来检测特定的内容。

实现方法

假设你有一幅图像，并且希望检测特定的内容（如在图 4.9 中检测空中的云）。首先要做的是选取感兴趣区域（Region of Interest, ROI），该区域包含目标物体的一份样本。这个区域在图 4.9 所示的测试截图中位于矩形框之内。

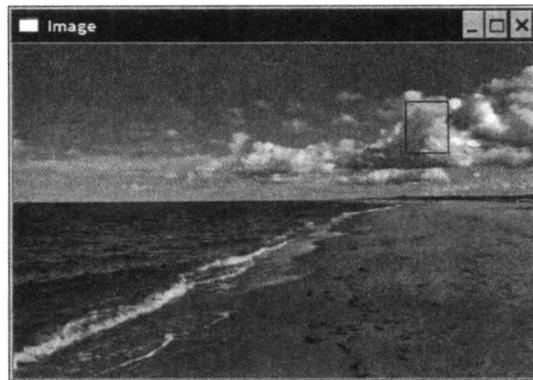


图 4.9

在我们的程序中，获取感兴趣区域的方法如下：

```
cv::Mat imageROI;
imageROI= image(cv::Rect(360,55,40,50)); // 包含云的区域
```

然后，提取该 ROI 的直方图。这可以通过使用本章第一则秘诀中定义的 Histogram1D 类得以实现：

```
Histogram1D h;
cv::MatND hist= h.getHistogram(imageROI);
```

通过归一化该直方图，我们得到一个函数，它给出了一个给定强度的像素术语这个区域的概率：

```
cv::normalize(histogram,histogram,1.0);
```

反投影直方图的作用是在于替换一个输入图像中每个像素值，使其变成归一化直方图中对应的概率值。

```
cv::calcBackProject(&image,
1,           // 一幅图像
channels,     // 通道数量
histogram,    // 进行反投影的直方图
result,       // 生成的反投影图像
ranges,       // 每个维度的值域
255.0         // 缩放因子
);
```

得到的结果即图 4.10 所示的概率图，概率从亮（低概率）到暗（高概率），表示像素属于参考区域的概率。

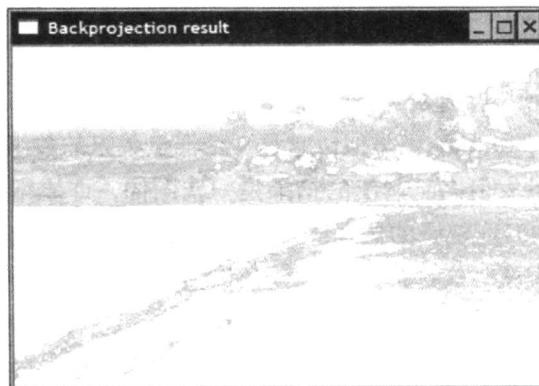


图 4.10

如果我们进行一次阈值化，可以获得最可能是“云”的像素，如图 4.11 所示。

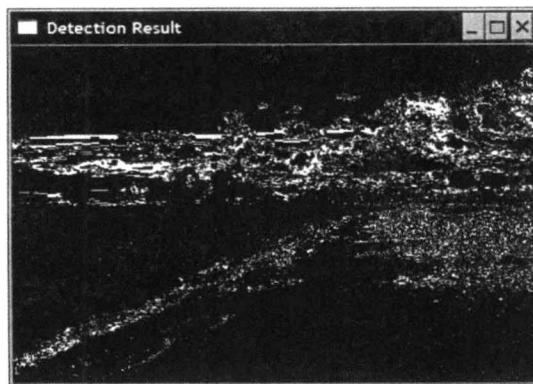


图 4.11

```
cv::threshold(result,result,255*threshold,  
255,cv::THRESH_BINARY);
```

作用原理

前面的结果也许会令人失望，因为云之外的其他区域也被错误地检测到。概率函数是从一张简单的灰度直方图中提取的，这一点很重要。图像中的许多其他像素与云层像素的强度是相同的，而在反投影直方图时相同强度的像素被替换成相同的概

率值。提高检测质量的一个方法是使用颜色信息。但是这么做意味着我们需要修改 cv:::calBackProject 的调用方式。

cv:::calBackProject 的用法与 cv:::calcHist 类似。第一个参数指定输入图像，然后需要列出使用到的通道数目。这一次，直方图变成了函数的一个参数。它需要预先被归一化，它的尺寸应该匹配通道列表数组的其中一项，以及输入的范围参数。后者的作用和 cv:::calcHist 中类似，它是一个浮点数组的数组，每一项指定了对于通道的范围（最小及最大值）。输出的结果是一幅图像，即概率映射图。由于每个像素都被替换成直方图中对应位置的概率值，生成图像的值都位于 0.0~1.0 之间（假设输入的一个归一化直方图）。最后一个参数允许你通过乘以一个因素重缩放这些值，这是一个可选项。

扩展阅读

现在让我们看看如何在直方图反投影算法中使用颜色信息。我们需要定义一个封装反投影过程的类，首先声明所需的类属性并进行初始化：

```
class ContentFinder{
private:
    float hranges[2];
    const float*ranges[3];
    int channels[3];
    float threshold;
    cv::MatND histogram;
public:
    ContentFinder():threshold(-1.0f){
        ranges[0]= hranges;// 所有通道的值相同
        ranges[1]= hranges;
        ranges[2]= hranges;
    }
}
```

接下来，我们定义一个阈值参数，将用于创建显示检测结果的二值映射图。如果这个参数设置为负值，返回的将是原始的概率映射图：

```
// 设置直方图的阈值 [0,1]
void setThreshold(float t){
```

```
    threshold=t;
}
// 获取阈值
float getThreshold(){
    return threshold;
}
```

输入的直方图必须是归一化后的：

```
// 设置参考直方图
void setHistogram(const cv::MatND& h){
    histogram= h;
    cv::normalize(histogram,histogram,1.0);
}
```

为了反投影直方图，你只需要指定图像、范围（我们假定所有的通道有着相同的范围）及所用通道的列表：

```
cv::Mat find(const cv::Mat& image,
             float minValue,float maxValue,
             int*channels,int dim) {
    cv::Mat result;
    hranges[0]= minValue;
    hranges[1]= maxValue;
    for(int i=0;i<dim;i++)
        this->channels[i]= channels[i];
    cv::calcBackProject(&image,1,// 输入图像
                        channels,           // 所用通道的列表
                        histogram,          // 直方图
                        result,              // 反投影的结果
                        ranges,              // 值域
                        255.0                // 缩放因子
    );
}
// 进行阈值化以得到二值图像
if(threshold>0.0)
    cv::threshold(result,result,
```

```
    255*threshold,255, cv::THRESH_BINARY);  
    return result;  
}
```

我们现在使用同一张图像的 BGR 直方图。这一次，将尝试检测蓝天所在区域。我们将首先读取彩色图像，并使用第 2 章中的减色函数，然后定义感兴趣区域：

```
ColorHistogram hc;  
// 读取彩图  
cv::Mat color= cv::imread("../waves.jpg");  
// 减色  
color= hc.colorReduce(color, 32);  
// 蓝天区域  
cv::Mat imageROI= color(cv::Rect(0,0,165,75));
```

接着，计算直方图并使用 find 方法以探测图片中的天空：

```
cv::MatND hist= hc.getHistogram(imageROI);  
ContentFinder finder;  
finder.setHistogram(hist);  
finder.setThreshold(0.05f);  
// 得到彩色直方图的反投影  
Cv::Mat result= finder.find(color);
```

彩色图像的检测结果显示如图 4.12 所示。

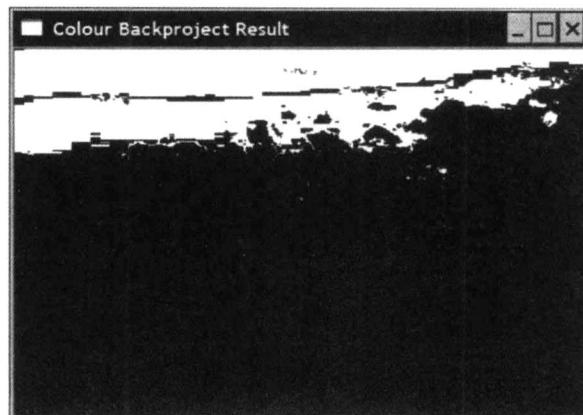


图 4.12

参 考

下一则秘诀将使用 HSV 颜色空间以探测到图像中的物体，这是另一个实现图像检测的解决方案。

4.6 使用均值漂移 (Mean Shift) 算法查找物体

反投影直方图的结果是一个概率映射，体现了已知的图像内容出现在图像中特定位置的概率。假设我们现在知道物体的近似位置，概率映射可用于找到的对象的确切位置。最有可能的位置是在已知窗口区域中得到最大概率的位置。因此，如果我们从最初的位置开始，并且迭代移动，便可以找到精确的位置。这便是均值漂移算法所要完成的任务。

实现方法

假设我们已经识别出了感兴趣的物体，如图 4.13 中狒狒的脸部（访问本书网站可以看到彩图）。



图 4.13

这次，我们将使用 HSV 颜色空间的色调（Hue）通道来描述物体。这意味着，需要转换图像到 HSV 空间并提取色调通道，然后计算特定 ROI 的 1D 色调直方图：

```
// 读取参考图像
cv::Mat image= cv::imread("../baboon1.jpg");
// 猴子脸部的 ROI
cv::Mat imageROI= image(cv::Rect(110,260,35,40));
// 获取色调通道的直方图
int minSat=65;
ColorHistogram hc;
cv::MatND colorhist=
    hc.getHueHistogram(imageROI,minSat);
```

可以看到，色调直方图可以通过 ColorHistogram 类中的一个添加的方法轻易获取：

```
// 使用掩码计算 1D 色调直方图
//BGR 图像需转换为 HSV 色彩空间
// 并去除低饱和度的像素
cv::MatND getHueHistogram(const cv::Mat &image,
                           int minSaturation=0) {
    cv::MatND hist;
    // 转换为 HSV 色彩空间
    cv::Mat hsv;
    cv::cvtColor(image,hsv,CV_BGR2HSV);
    // 是否使用掩码
    cv::Mat mask;
    if(minSaturation>0) {
        // 分割三通道为三幅图像
        std::vector<cv::Mat>v;
        cv::split(hsv,v);
        // 标出低饱和度的像素
        cv::threshold(v[1],mask,minSaturation,255,
                      cv::THRESH_BINARY);
    }
    //1D 色调直方图的参数
    hranges[0]= 0.0;
    hranges[1]= 180.0;
    channels[0]= 0;// 色调通道
    // 计算直方图
```

```
cv::calcHist(&hsv,
    1,           // 仅计算一幅图
    channels,    // 通道数量
    mask,        // 二值掩码
    hist,        // 返回的直方图
    1,           // 这是 1D 的直方图
    histSize,    // 项的数量
    ranges       // 像素值的范围
);
return hist;
}
```

产生的直方图之后作为 ContentFinder 实例的参数：

```
ContentFinder finder;
finder.setHistogram(colorhist);
```

现在，我们打开第二张图像，希望在这里找到新的狒狒脸部的位置。这幅图像需要转换到 HSV 空间：

```
image= cv::imread("../baboon3.jpg");
// 显示图像
cv::namedWindow("Image 2");
cv::imshow("Image 2",image);
// 转到到 HSV 空间
cv::cvtColor(image,hsv,CV_BGR2HSV);
// 分割图像
cv::split(hsv,v);
// 识别低饱和度的像素
cv::threshold(v[1],v[1],minSat,255,cv::THRESH_BINARY);
```

接下来，我们使用之前获取的直方图获取这张图像的色调通道的反投影：

```
// 获取直方图的反投影
result= finder.find(hsv,0.0f,180.0f,ch,1);
// 去除低饱和度的像素
cv::bitwise_and(result,v[1],result);
```

现在，从最初的矩形区域（即狒狒脸在最初的图像中的位置）开始，

cv::meanShift 算法将会在新的位置更新这个矩形对象：

```
cv::Rect rect(110,260,35,40);
cv::rectangle(image,rect,cv::Scalar(0,0,255));
cv::TermCriteria criteria(cv::TermCriteria::MAX_ITER,
                           10,0.01);
cv::meanShift(result,rect,criteria);
```

最初的脸部和新的脸部位置都显示在图 4.14 的截图中。

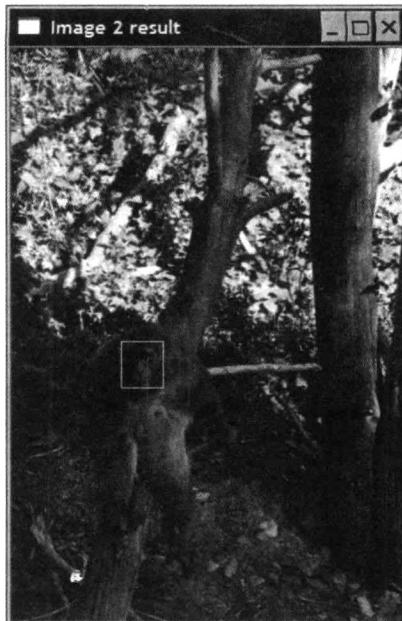


图 4.14

作用原理

在这个例子中，我们使用 HSV 颜色空间中的色调分量以描述所要搜索的物体。因此，图像必须先被转换。当使用 CV_BGR2HSV 标志时，色调分量位于生成的图像的第一个通道。这个分量是 8 位的，值的范围为 0~180 (cv::cvtColor 转换的图像的类型与输入图像一致)。为了提取色调图像，使用 cv::split 函数将三通道的 HSV 图像分割为三个单通道的图像，这三个图像被置入 std::vector 对象中，而色调图像是该向量的第一项，位于索引 0。

当使用一个颜色的色调分量时,考虑它的饱和度分量也是很重要的(向量的第2项)。事实上,如果一个颜色的饱和度偏低,会导致色调信息变得不稳定以及不可靠。这是因为低饱和度的颜色中,红绿蓝三个分量几乎是相等的。这使得难以确定精确的颜色。因此,我们决定忽略低饱和色的色调分量。也就是说,在直方图中不统计它们的数目(方法 `getHueHistogram` 中的参数 `minSat` 用于剔除饱和度低于该阈值的像素),并且在反投影的结果中也不考虑它们(在调用 `cv::meanShift` 前使用 `cv::bitwise_and` 操作符能够移除检测到的所有低饱和度像素)。

均值漂移算法以迭代的方式锁定概率函数的局部最大值。它的原理是寻找预定义窗口中数据点的重心,或者说加权平均值。该算法将窗口中心移动到数据点的重心处,并重复这个过程直到窗口重心收敛到一个稳定点。OpenCV 文档定义了两种终止条件:迭代的最大次数以及窗口中心的位移值(低于该值即认为算法已经收敛)。这两个条件保存在 `cv::TermCriteria` 实例中。`cv::meanShift` 函数返回本次调用中迭代的次数。显而易见,结果的质量取决于输入的概率图及初始位置。

参 考

均值漂移算法在视觉跟踪中大量使用。第 10 章将详细讨论物体跟踪的问题。

OpenCV 还实现了 CamShift 算法,这是一个改进版本的均值漂移算法,搜索窗口的尺寸和朝向会发生改变。

4.7 通过比较直方图检索相似图片

基于内容的图像检索是计算机视觉中的一个重要问题。它的目标是给定一个查询图像,寻找呈现内容相似的一组图像。由于我们已经了解到直方图能够有效地描述图像的内容,可以认为它们也能用于解决基于内容的检索问题。

这里的关键点在于通过简单地比较它们的直方图来测量两个图像的相似性,需要定义一个评估两个直方图多么不同、或者多么相似的测量函数。人们已经提出了多种这样的测量方法,而 OpenCV 在 `cv::compareHist` 函数中实现了其中一些。

实现方法

为了将一组图像与参考图像作比较,找到那些最像查询的图像的,我们创建了

ImageComparator 类。它包含对一个查询图像和输入图像的引用，连同它们的直方图（cv::MatND 实例）。此外，因为我们使用颜色直方图进行比较，还用到了 ColorHistogram 类：

```
class ImageComparator{
private:
    cv::Mat reference;
    cv::Mat input;
    cv::MatND refH;
    cv::MatND inputH;
    ColorHistogram hist;
    int div;
public:
    ImageComparator():div(32){
    }
}
```

为了获得一个可靠的相似度度量，必须降低颜色的数量。因此，该类包含一个减色因子，可用于查询图像和输入图像：

```
// 减色因子
// 比较的将是减色后的图像
// 色彩空间中的每个维度都将按照该变量进行减色
void setColorReduction(int factor){
    div= factor;
}
int getColorReduction(){
    return div;
}
```

查询图像使用一个适当的赋值函数进行指定，同时也减少图像的颜色：

```
void setReferenceImage(const cv::Mat& image) {
    reference= hist.colorReduce(image,div);
    refH= hist.getHistogram(reference);
}
```

最后，用一个比较函数比较参考图像与给定的输入图像。该方法返回一个分数表明这两个图像的相似度：

```

double compare(const cv::Mat& image) {
    input= hist.colorReduce(image,div);
    inputH= hist.getHistogram(input);
    return cv::compareHist(
        refH,inputH,CV_COMP_INTERSECT);
}
};

```

这个类可用于检索与一个给定的查询图像相似的图像。查询图像的设置方法如下：

```

ImageComparator c;
c.setReferenceImage(image);

```

在这里，我们使用的查询图像是秘诀“反投影直方图以检测特定的图像内容”中的海滩图像的彩色版本。这幅图像与图 4.15 所示的图像进行比较，图像的显示顺序是从相似度最高的到最低的。

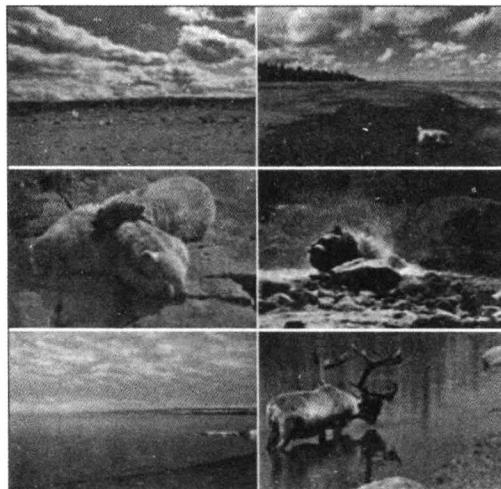


图 4.15

作用原理

大多数直方图比较是基于逐个容器进行的比较时，即比较直方图容器时并不考虑相邻容器的影像。因此，测量相似度之前减少颜色空间是很重要的。其他颜色空间也可以这么使用。

调用 `cv::compareHist` 很直接明了，只需提供两个直方图，函数便返回测量距离。通过一个标志参数可以指定测量方法。在 `ImageComparator` 类使用的是交叉测量法（参数为 `CV_COMP_INTERSECT`）。该方法简单比较每个直方图容器的值，并保留最小的一个。相似性测量值只是这些最小值的和。因此，如果两个图像的直方图没有共同的颜色将得交叉值 0，而两个相同的直方图的返回值等于像素的总数。

其他可用的方法包括对容器间插值的归一化平方进行求和的卡方检验（参数为 `CV_COMP_CHISQR`），信号处理中的归一化互相关方法（参数为 `CV_COMP_CORREL`）以及统计学中用于评估两个概率分布间相似性的巴氏距离测量（参数为 `CV_COMP_BHATTACHARYYA`）。

参 考

OpenCV 的文档给出了各种直方图比较方法的确切公式。

挖土机的距离（Earth Mover Distance）是另一种流行的直方图比较方法，这种方法的主要优势是考虑了直方图中相邻容器的值。提出这个概念的论文是 “*The Earth Mover's Distance as a Metric for Image Retrieval*” (Y.i Rubner, C.Tomasi, L.J.Guibas, *Int.Journal of Computer Vision*, Vol.40, No.2, 2000, pp.99-121)。

第 5 章

基于形态学运算的图像变换

本章主要探讨：

- ◆ 使用形态学滤波对图像进行腐蚀、膨胀运算；
- ◆ 使用形态学滤波对图像进行开闭运算；
- ◆ 使用形态学滤波对图像进行边缘及角点检测；
- ◆ 使用分水岭算法对图像进行分割；
- ◆ 使用 GrabCut 算法提取前景物体。

5.1 引言

形态学滤波理论于上世纪 90 年代提出，被用于分析及处理离散图形。它定义了一系列的运算，应用预定义的形状元素来变换一张图像。形状元素与像素相邻点相交的方式确定了运算的结果。本章介绍最重要的形态学运算，同时探讨如何使用形态学算法进行图像分割的问题。

5.2 使用形态学滤波对图像进行腐蚀、膨胀运算

腐蚀和膨胀是最基本的形态学运算。因此，我们将在第一则秘诀中介绍它们。数学形态学中最基本的工具是结构元素。结构元素简单地定义为像素的结构（形状），以及一个原点（又称为锚点）。使用形态学滤波涉及对图像的每个像素应用这个结构元素。当结构元素的原点与给定的像素对齐时，它与图像的相交部分定义了一组进行形态学运算的像素。原则上说，结构元素可以是任何形状，但通常使用简单的形状，例如方形、圆形或菱形，而原点位于中心位置（基于对效率的考虑）。

准备工作

由于形态学滤波通常使用于二值图像，我们将借助前一章第一则秘诀中使用的阈值化技术生成二值图像。由于形态学的惯例是用高（白色）像素表示前景物体，用低（黑色）像素表示背景，我们还需要对图像取反。在形态学术语中，图 5.1 被称为前一章生成的图片的补（Complement）。

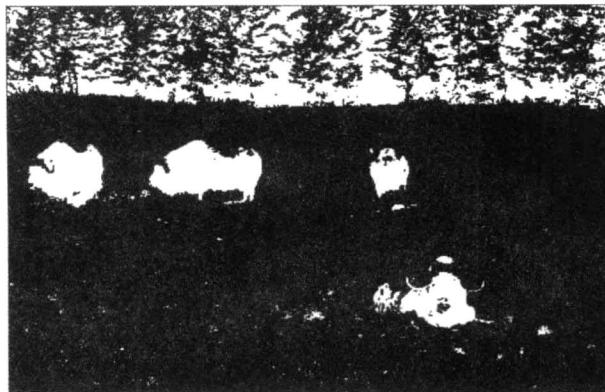


图 5.1

实现方法

在 OpenCV 中，腐蚀和膨胀由函数 `cv::erode` 及 `cv::dilate` 实现，用法很直接：

```
// 读取输入图像
cv::Mat image= cv::imread("binary.bmp");
// 腐蚀图像
cv::Mat eroded;      // 目标图像
cv::erode(image,eroded,cv::Mat());
// 显示腐蚀后的图像
cv::namedWindow("Eroded Image");
cv::imshow("Eroded Image",eroded);
// 膨胀图像
cv::Mat dilated;      // 目标图像
cv::dilate(image,dilated,cv::Mat());
// 显示膨胀后的图像
cv::namedWindow("Dilated Image");
cv::imshow("Dilated Image",dilated);
```

这些函数生成的两张图像显示在下面的截图中。首先显示的是腐蚀，如图 5.2 所示。

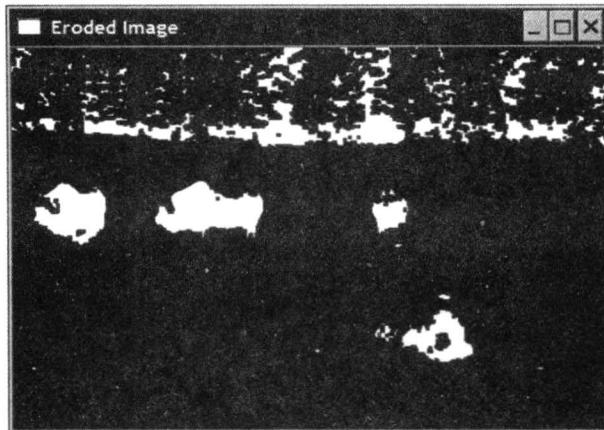


图 5.2

然后是膨胀的结果，如图 5.3 所示。

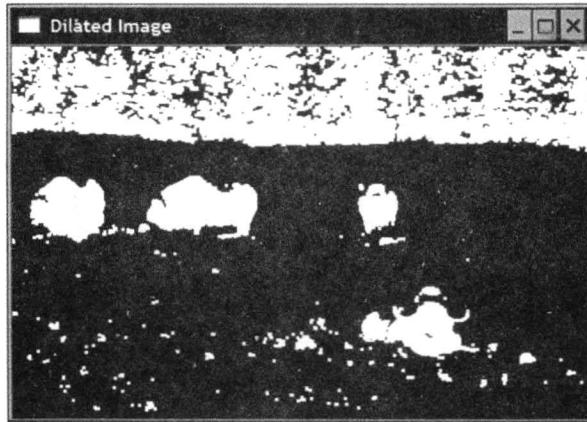


图 5.3

作用原理

与所有其他的形态学滤波一样，本秘诀中的两个滤波也运算在每个像素周围的像素集合（即邻居）上，这是由结构元素定义的。当应用到一个给定像素时，结构元素的锚点与该像素的位置对齐，而所有与它相交的像素都被包括在当前的像素集合中。腐蚀替换当前像素位像素集合中找到的最小像素值。膨胀则是相反的运算，它替换当前像素位像素集合中找到的最大像素值。由于输入的二值图像仅包含黑色（0）和白色

(255) 像素，每个像素只会被替换为白色像素或黑色像素。

要想象出两个运算的效果有个好办法，就是考虑背景（黑色）和前景（白色）物体。腐蚀的情况下，如果给定像素位置的结构元素接触到背景（即，相交集合中的一个像素为黑色），那么该像素将被设置为背景。而在膨胀的情况下，如果给定像素位置的结构元素接触到前景物体，那么该像素将被设置为白色。这解释了为何在腐蚀后的图像中，物体的尺寸会减小。可以观察到一些非常细小的物体（可被认为是背景像素中的“噪音”）被完全移除了。相似的，膨胀后的物体更大，同时内部的一些“洞”被填满。

OpenCV 默认使用 3×3 的方形结构元素。当函数的第 3 个参数被设置为空矩阵（即 `cv::Mat()`）时，这个默认的结构元素被使用，之前的例子便这么使用的。你也可以通过一个矩阵参数指定结构元素的尺寸（与形状），矩阵中的非零项定义了结构元素。在下例中，使用了一个 7×7 的结构元素。

```
cv::Mat element(7,7,CV_8U,cv::Scalar(1));
cv::erode(image,eroded,element);
```

在这个例子中的效果更有毁灭性，如图 5.4 所示。

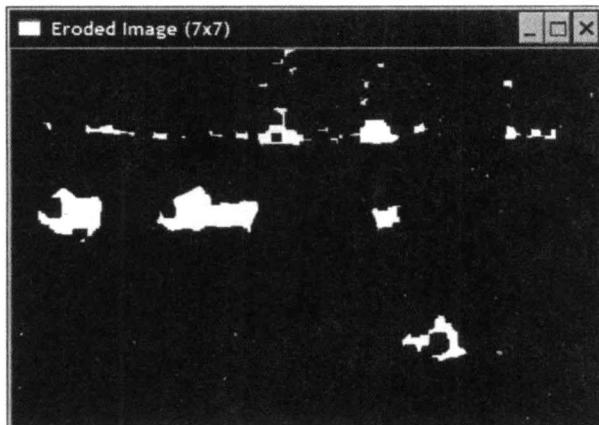


图 5.4

另一个获得相同结果的方法是对一幅图像反复应用同一个结构元素。这两个函数都有一个可选的参数用于指定重复的次数：

```
// 腐蚀图像三次
cv::erode(image,eroded, cv::Mat(), cv::Point(-1,-1), 3);
```

原点参数为 `cv::Point(-1, -1)` 意味着原点位于矩阵的中心（这是默认参数），可以被修改为结构元素中任意位置，获得的图像与通过 7×7 结构元素的结果是一致的。事实上，腐蚀一幅图像两次就像是让结构元素对自己进行膨胀后回去腐蚀同一幅图像。这点对于膨胀运算也适用。

最后，由于背景 / 前景的概念是任意的，我们可以观察下述结论（这是腐蚀 / 膨胀运算的一个基本属性）。使用一个结构元素来腐蚀背景物体也可以视为对图像背景部分的膨胀运算。或者更正式的说法：

- ◆ 对图像的腐蚀运算等于对图像负片的膨胀运算的负片；
- ◆ 对图像的膨胀运算等于对图像负片的腐蚀运算的负片。

扩展阅读

这里，我们的形态学滤波都作用在二值图像上，但是相同定义的运算也可以用于灰度图像。

OpenCV 的形态学函数支持原地处理。这意味着，你的输入图形与目标图像可以是同一个，所以能这么写：

```
cv::erode(image, image, cv::Mat());
```

OpenCV 在实现中会创建所需的临时图像。

参 考

下一则秘诀将串联使用腐蚀与碰撞以生成新的运算。

“使用形态学滤波对图像进行边缘及角点检测” 中将介绍灰度图像中形态学滤波的应用。

5.3 使用形态学滤波对图像进行开闭运算

之前的秘诀绍了两个基本的形态学运算：腐蚀与碰撞。由它们可以定义出其他的运算，下面的两则秘诀将展示其中的一些。开闭运算将是本秘诀的内容。

实现方法

为了使用更高级的形态学滤波，你需要通过合适的参数调用 `cv::morphologyEx`

函数。例如，下面执行的便是闭运算：

```
cv::Mat element5(5,5,CV_8U, cv::Scalar(1));
cv::Mat closed;
cv::morphologyEx(image,closed, cv::MORPH_CLOSE,element5);
```

这里使用的 5×5 的结构元素，使得滤波的效果更明显。如果我们输入的图像是先前秘诀中的二值图像，可以得到图 5.5 所示的图像。

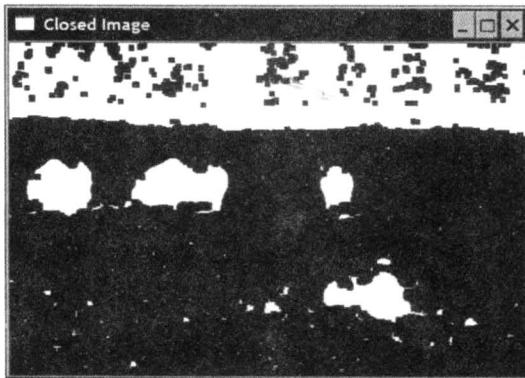


图 5.5

类似地，应用形态学开运算将产生图 5.6 所示的图像。

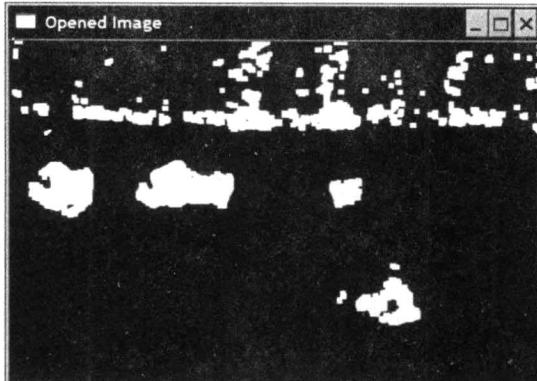


图 5.6

生成的图像来源于下述代码：

```
cv::Mat opened;
cv::morphologyEx(image,opened, cv::MORPH_OPEN,element5);
```

作用原理

用基本的腐蚀和膨胀运算可以定义开、闭滤波器：

- ◆ 闭运算定义为对图像先膨胀，再腐蚀；
- ◆ 开运算定义为对图像先腐蚀，再膨胀。

因此，可以使用下述调用计算一幅图像的闭：

```
// 膨胀原图  
cv::dilate(image,result, cv::Mat());  
// 对膨胀后的图像进行原地腐蚀  
cv::erode(result,result, cv::Mat());
```

开运算的结果通过颠倒两个函数的次序即可得到。

在检验闭滤波器的结果时，可以看到白色前景物体中的小洞被填充。该滤波同时连接多个相邻物体。基本上，无法完全包含结构元素的洞或缝隙都将被滤波移除。

反过来，开滤波器移除的是场景中的比较小的物体，因为它们无法完全包含结构元素。

这些滤波器通常在物体检测中使用。闭滤波器将误分割成碎片的物体重新连接，而开滤波器图像噪点引起的小像素块（Blob）。因此，在视频序列中使用它们很有帮助。如果我们用于测试的二值图像相继进行闭、开运算，获得的图像将只显示场景中的主要物体，如图 5.7 所示。如果你优先处理噪点，你也可以先进行开运算，再进行闭运算，但是有可能去除一些分散的物体。

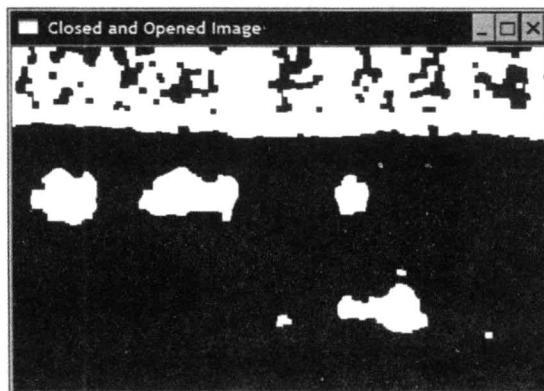


图 5.7

需要注意的是，对一幅图像多次使用相同的开运算（或者闭运算）时是没有效果的。事实上，在第一次的开运算填充了图像中的洞后，再次应用相同的滤波不会对图像产生任何变化。用数学的术语讲，这些运算是等幂的（Idempotent）。

5.4 使用形态学滤波对图像进行边缘及角点检测

形态学滤波也可用于检测图像中指定的特征。在这则秘诀中，我们将学习如何检测灰度图中的直线和角点。

准备工作

本秘诀将使用图 5.8 所示的图像。



图 5.8

实现方法

首先定义名为 MorphoFeatures 的类，我们将使用它来检测图像特征：

```
class MorphoFeatures{
    private:
        // 用于生成二值图像的阈值
        int threshold;
        // 角点检测中用到的结构元素
```

```
cv::Mat cross;
cv::Mat diamond;
cv::Mat square;
cv::Mat x;
```

使用 cv::morphologyEx 函数配上合适的滤波器可以轻松实现直线的检测：

```
cv::Mat getEdges(const cv::Mat &image) {
    // 得到梯度图
    cv::Mat result;
    cv::morphologyEx(image, result,
                     cv::MORPH_GRADIENT, cv::Mat());
    // 阈值化以得到二值图像
    applyThreshold(result);
    return result;
}
```

一个简单的私有函数用于获取二值的边缘图像：

```
void applyThreshold(cv::Mat& result) {
    // 使用阈值化
    if(threshold>0)
        cv::threshold(result, result,
                      threshold, 255, cv::THRESH_BINARY);
}
```

在 main 函数中使用该类，可以得到边缘图像：

```
// 创建形态学特征实例
MorphoFeatures morpho;
morpho.setThreshold(40);
// 获取边缘
cv::Mat edges;
edges=morpho.getEdges(image);
```

结果如图 5.9 所示。



图 5.9

使用形态学检测角点要复杂些，因为 OpenCV 没有直接实现它。这是一个很好的使用非方形结构元素的例子。事实上，它需要定义四种不同的结构元素，包括方形、菱形、十字形以及 X 形，这都是在构造函数中完成的（简单起见，所有这些元素的尺寸都固定为 5×5 ）：

```
MorphoFeatures():threshold(-1),
    cross(5,5,CV_8U,cv::Scalar(0)),
    diamond(5,5,CV_8U,cv::Scalar(1)),
    square(5,5,CV_8U,cv::Scalar(1)),
    x(5,5,CV_8U,cv::Scalar(0)) {
    // 创建十字形元素
    for(int i=0;i<5;i++) {
        cross.at<uchar>(2,i)= 1;
        cross.at<uchar>(i,2)= 1;
    }
    // 创建菱形元素
    diamond.at<uchar>(0,0)= 0;
    diamond.at<uchar>(0,1)= 0;
    diamond.at<uchar>(1,0)= 0;
    diamond.at<uchar>(4,4)= 0;
    diamond.at<uchar>(3,4)= 0;
    diamond.at<uchar>(4,3)= 0;
```

```
diamond.at<uchar>(4,0)= 0;
diamond.at<uchar>(4,1)= 0;
diamond.at<uchar>(3,0)= 0;
diamond.at<uchar>(0,4)= 0;
diamond.at<uchar>(0,3)= 0;
diamond.at<uchar>(1,4)= 0;
// 创建 X 形元素
for(int i=0;i<5;i++) {
    x.at<uchar>(i,i)= 1;
    x.at<uchar>(4-i,i)= 1;
}
}
```

在检测角点特征的过程中，需要接连使用这些结构元素以得到最终的角点映射图：

```
cv::Mat get Corners(const cv::Mat &image) {
    cv::Mat result;
    // 十字形膨胀
    cv::dilate(image,result,cross);
    // 菱形腐蚀
    cv::erode(result,result,diamond);
    cv::Mat result2;
    // X 形膨胀
    cv::dilate(image,result2,x);
    // 方形腐蚀
    cv::erode(result2,result2,square);
    // 通过对两张图像做差值得到角点图像
    cv::absdiff(result2,result,result);
    // 阈值化以得到二值图像
    applyThreshold(result);
    return result;
}
```

为了更好地可视化检测的结果，使用下述方法在二值图像中的每个检测点上绘制一个圆：

```
void drawOnImage(const cv::Mat& binary,
                  cv::Mat& image){
    cv::Mat<uchar>::const_iterator it=
```

```
binary.begin<uchar>();
cv::Mat<uchar>::const_iterator itend=
    binary.end<uchar>();
// 遍历每个像素
for(int i=0;it!= itend;++it,++i){
    if(!*it)
        cv::circle(image,
                   cv::Point(i%image.step,i/image.step),
                   5, cv::Scalar(255,0,0));
}
}
```

使用下述代码检测图像中的角点：

```
// 得到角点
cv::Mat corners;
corners= morpho.getorners(image);
// 在图像中显示角点
morpho.drawOnImage(corners,image);
cv::namedWindow("Corners on Image");
cv::imshow("Corners on Image",image);
```

图 5.10 便是检测后的图像。

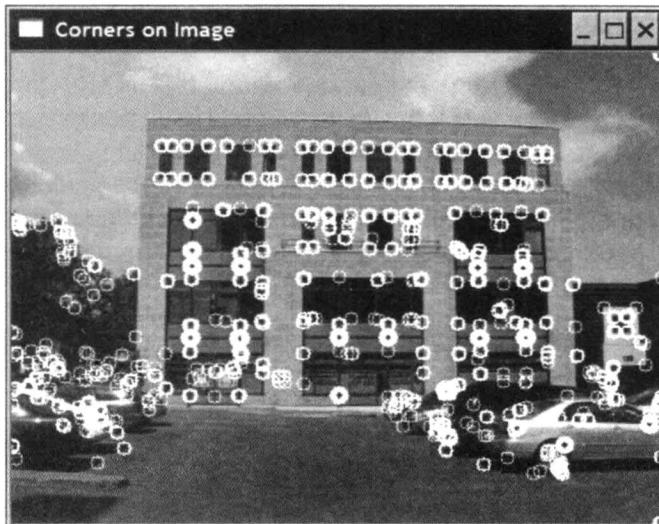


图 5.10

作用原理

为了更好地理解形态学运算在灰度图上的效果，可以将图像的灰度视作海拔高度。明亮的区域对应的是山岭，而灰暗的区域则是山谷。同时，由于边缘意味着明亮与灰暗像素的快速过渡，可以视作陡峭的悬崖。如果腐蚀运算应用于这么一个地形，结果将是用相邻范围内的最小值来替换每个像素，于是降低了它的高度。悬崖将“腐蚀”，而山谷被延伸。膨胀的结果恰恰相反，山谷之间将竖起很多悬崖。然而，不论是哪种情况，平原（即强度不变的区域）相对而言保持原样。

由上述的观察可以引出检测图像边缘（或悬崖）的一种简单方法，即计算膨胀后的图像与腐蚀后的图像的差值。由于这两个变换后的图像不同的地方主要在边缘处，图像边缘将通过求差得到强化。这正是 `cv::morphologyEx` 在参数为 `cv::MORPH_GRADIENT` 的作用原理。结构元素的尺寸越大，检测出的边缘越厚。边缘检测运算又被称为 Beucher 梯度（下一章节将详细介绍梯度）。通过将膨胀后的图像减去原图，或是将原图减去腐蚀后的图像，都可以得到类似结果，不过得到的边缘将薄很多。

角点检测要复杂些，因为它使用四种不同的结构元素。OpenCV 并没有实现该运算，我们用它来展示如何定义及组合不同形状的结构元素。背后的原理是使用膨胀来进行闭运算，并使用两种不同的结构元素来应用腐蚀运算。挑选的这些元素可以确保连续的边缘不受改变，但是由于它们的重复效应，角点处的边缘仍受影响。图 5.11 所示的简单图像由单个白色方形组成，方便我们更好地理解闭运算的非对称结果。

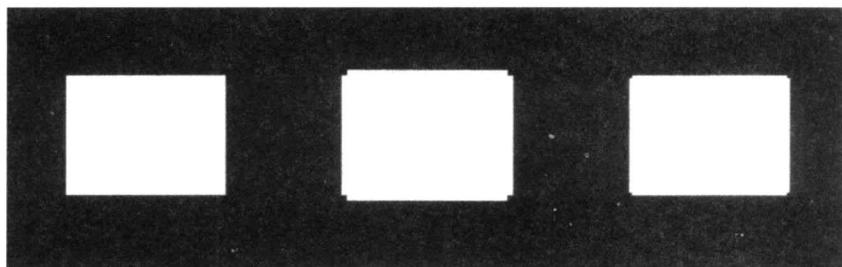


图 5.11

第一个方块是原图。在被十字形元素膨胀后，方块的边缘被扩张，而由于十字形元素没有击中角点，此处不受影响。中间的方块描述的便是这个结果。膨胀后的图像接着被菱形元素腐蚀。这次运算将大多数的边缘恢复到原始位置，但是之前没有膨胀过的角点被向内推动。之后得到了左边的方块，可以看到，它缺少明显的角点。同样的处理过程通过 X 形与方形元素得到重复。这两个元素是先前元素的旋转版本，捕获

的将是 45° 旋转后的角点。最后，对两次过程的结果做差值，提取出角点特征。

参 考

论 文 “*Morphological gradients*” (J.-F.Rivest, P.Soille, S.Beucher, *ISET's symposium on electronic imaging science and technology*, SPIE, 1992.2) 中有更多关于形态学梯度的介绍。

阅读论文 “*A modified regulated morphological corner detector*” (F.Y.Shih, C.-F.Chuang, V.Gaddipati, *Pattern Recognition Letters*, Vol.26, No.7, 2005.5)，以获取形态学角点检测方面的更多信息。

5.5 使用分水岭算法对图像进行分割

分水岭变换是一个流行的图像处理算法，用于快速分割图像为同类区域。它背后的原理是，如果将图像视为拓扑结构的地图，那么均质区域对应的是被陡峭边缘包围的平坦盆地。由于这个算法过于简单，最初版本极有可能将图像过度分割为多个微小区域。因此 OpenCV 提供了该算法的变种，使用预定义的一组标记来引导对图像的分割。

实现方法

分水岭分割的结果通过 `cv::watershed` 函数获取。函数的输入是一个 32 位的有符号整数标记图，每个非零的像素表示一个标记。我们将图像中已知属于某个区域的像素进行标记。基于这个初始标记，分水岭算法开始确定其他像素的归属区域。在该秘诀中，我们将首先创建灰度格式的标记图，然后将它转换为一张整数图。这些步骤都被封装在 `WatershedSegmenter` 类中：

```
class WatershedSegmenter{
private:
    cv::Mat markers;
public:
    void setMarkers(const cv::Mat& markerImage) {
        // 转换为整数图像
        markerImage.convertTo(markers,CV_32S);
    }
}
```

```
cv::Mat process(const cv::Mat &image) {
    // 使用算法
    cv::watershed(image, markers);
    return markers;
}
```

这些标记获取的方式取决于应用程序。例如，可以通过预处理过程识别出属于目标物体的像素。分水岭算法将通过最初的检测明确划分出完整的物体。在该秘诀中，我们简单使用二值图像来标识对应的原始图像中的动物（这是第4章开始时显示的图像）。

因此，从二值图像，我们需要识别出确定属于前景（动物）的像素，以及确定术语背景（主要是草地）的像素。我们将用标签255标记前景像素，用标签128标记背景像素（挑选的方式是完全任意的，不同于255的数都可行）。其他的像素，它们的标签是未知的，被赋值为0。现在，二值图像包含了过多的白色像素，它们属于不同部分。通过大幅度地腐蚀该图像可以获取仅仅属于重要物体的像素：

```
// 移除噪点与微小物体
cv::Mat fg;
cv::erode(binary, fg, cv::Mat(), cv::Point(-1,-1), 6);
```

结果如图5.12所示。

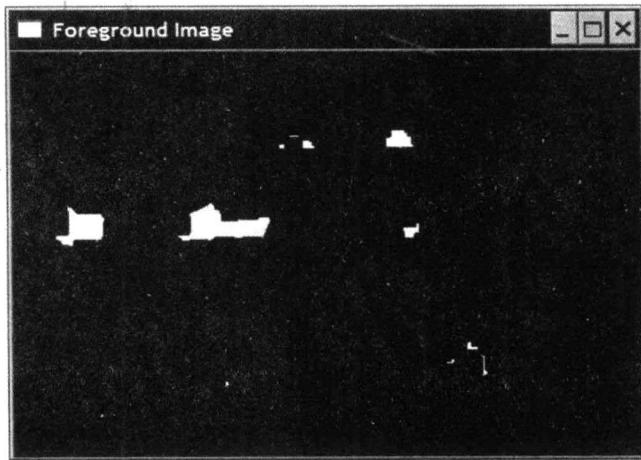


图 5.12

可以看到属于背景森林的一些像素依然存在，简单地保留它们。它们将认为是目标物体的一部分。相似地，我们也通过原始二值图像的大幅度膨胀获取属于背景的元素：

```
// 识别不包含物体的像素  
cv::Mat bg;  
cv::dilate(binary, bg, cv::Mat(), cv::Point(-1,-1), 6);  
cv::threshold(bg, bg, 1, 128, cv::THRESH_BINARY_INV);
```

生成的黑色像素对应的是背景像素。这就是膨胀之后的阈值化运算将像素赋值为 128 的原因。于是，得到图 5.13。

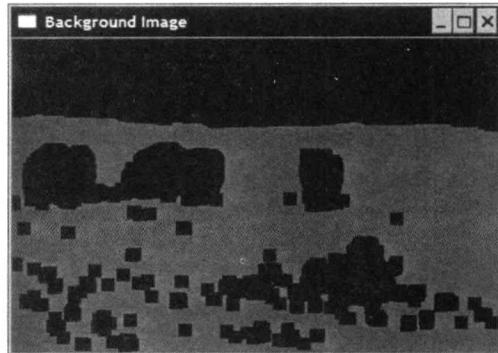


图 5.13

组合使用这些图可以形成标记图形：

```
// 创建标记图像  
cv::Mat markers(binary.size(), CV_8U, cv::Scalar(0));  
markers= fg+bg;
```

例子中使用重载后的操作符 + 组合两幅图像，这将是分水岭算法的输入参数，如图 5.14 所示。

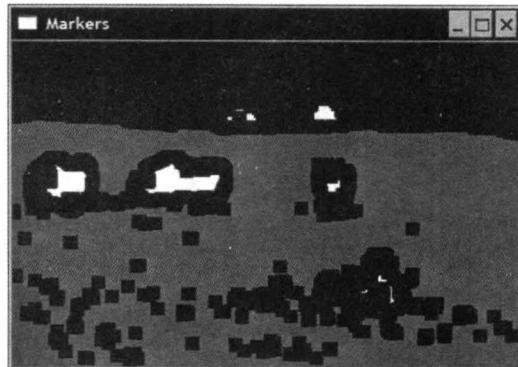


图 5.14

分割的结果以如下方式获取：

```
// 创建分水岭分割对象  
WatershedSegmenter segmenter;  
// 设置标记，并进行处理  
segmenter.setMarkers(markers);  
segmenter.process(image);
```

标记图像得到更新，每个零像素被赋值为输入标签之一，而隶属于当前发现的边界的像素被赋予 -1。结果，标记图像如图 5.15 所示。

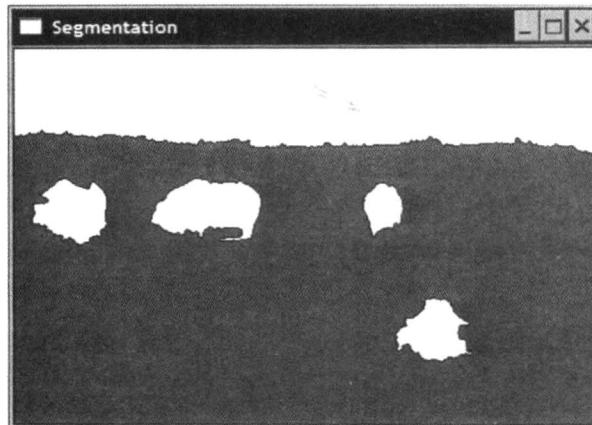


图 5.15

边界图像如图 5.16 所示。

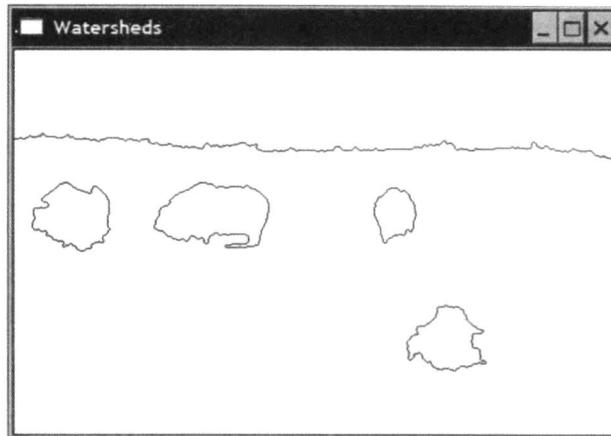


图 5.16

作用原理

正如我们在之前秘诀中所做的，在描述分水岭算法时我们也使用形态学地图做类比。为了创建分水岭分割，我们逐渐地在第0层填充图像。随着“水”逐渐涨高（到第1、2、3层等等），蓄水的盆地形成了。这些盆地的尺寸也缓缓增加，以至于最终两个不同盆地的水汇聚。当这发生时，创建一个分水岭以保持两个盆地分离。一旦水的层数到达它的最大值，这些创建的盆地和分水岭的集合形成了分水岭分割算法的结果。

不难预料，在注水过程刚开始时会创建许多小型的单个盆地。当所有这些盆地的水汇聚时，创建了许多分水岭，这导致了过度分割的图像。为了解决这个问题，算法得到修改，注水的过程从一组标记像素集合开始。由这些标记创建的盆地所赋的值与初始的标记保持一致，当标签相同的两个盆地汇聚时，不生成分水岭，避免了过度分割。

这便是调用 `cv::watershed` 函数时所发生的事情。输入的标记图像得到更新，生成最终的分水岭分割。标记图像可以拥有任意数量的标签，未知像素的标签被赋予 0。标记图像的类型是 32 位的有符号整数，这是为了能够定义超过 255 种标签。特殊的值 -1 也被允许赋值给属于某个分水岭的像素。这正是 `cv::watershed` 函数返回的格式。为了简化结果的显示，我们还定义了两个特殊方法。第一个方法返回标记图像（分水岭位于值 0）。这通过阈值化实现：

```
// 以图像的形式返回结果
cv::Mat getSegmentation(){
    cv::Mat tmp;
    // 标签高于 255 的分割一律赋值为 255
    markers.convertTo(tmp,CV_8U);
    return tmp;
}
```

类似地，第二个方法返回的图像中分水岭上的像素被赋值为 0，剩下的都为 255。这回 `cv::convertTo` 方法用于实现以下结果：

```
// 以图像的形式返回分水岭
cv::Mat getWatersheds(){
    cv::Mat tmp;
    // 转换前每个像素都为 255p+255
    markers.convertTo(tmp,CV_8U,255,255);
    return tmp;
}
```

格式转化之前的线性变换允许 -1 像素转化为 0 (因为 $-1 \times 255 + 255 = 0$)。大于 255 的像素被赋值为 255，这是因为有符号整数转化为无符号字符时会进行饱和运算。

参 考

要想了解更多分水岭的信息，请阅读文章 “*The Viscous Watershed Transform*” (C. Vachier, F.Meyer, *Journal of Mathematical Imaging and Vision*, Vol.22, No.2-3, 2005.5.)。

下一则秘诀展示了另一种图像分割算法，它同时能够将一幅图像分割为背景及前景物体。

5.6 使用GrabCut算法提取前景物体

OpenCV 实现了另一个流行的图像分割算法：GrabCut 算法。它并非基于数学形态学，放在这一章的目的是展示它与分水岭算法的相似之处。GrabCut 在计算时更加复杂，但是产生的结果要精确许多。对于期望从静态图像中提取前景物体的应用，如将一幅图中的物体剪贴到另一幅图中，这是最佳算法。

实现方法

`cv::grabCut` 函数使用方便，你只需提供图像并标记出背景像素及前景像素。基于局部的标记，算法将确定完整图像中的前景与背景分割。一种指定局部标记的方式是定义一个矩形，前景物体被它包围：

```
// 打开图像
image= cv::imread("../group.jpg");
// 定义前景物体的包围盒
cv::Rect rectangle(10,100,380,180);
```

矩形之外的所有像素都将标记为背景。除了输入的图像和分割图像，调用 `cv::grabCut` 函数还需要定义两个矩阵，它们将包含算法创建的模型：

```
cv::Mat result;           // 分割 (四种可能的值)
cv::Mat bgModel,fgModel; // 模型 (供内部使用)
//GrabCut 分割
cv::grabCut(image,        // 输入图像
```

```
    result,          // 分割结果
    rectangle,       // 包含前景物体的矩形
    bgModel,fgModel, // 模型
    5,              // 迭代次数
    cv::GC_INIT_WITH_RECT); // 使用矩形进行初始化
```

在最后一个参数中指定 `cv::GC_INIT_WITH_RECT` 表示我们使用的是包围盒模式（之后将讨论其他可选模式）。输入 / 输出分割图像可以有四种数值：

- ◆ `cv::GC_BGD`: 确定属于背景的像素（如本例中矩形之外的像素）；
- ◆ `cv::GC_FGD`: 确定属于前景的像素（本例不包含这种像素）；
- ◆ `cv::GC_PR_BGD`: 可能属于背景的像素；
- ◆ `cv::GC_PR_FGD`: 可能属于前景的像素（如本例中矩形之内的像素）。

通过提取数值等于 `cv::GC_PR_FGD` 的像素，我们得到二值的分割图像：

```
// 得到可能为前景的像素
cv::compare(result, cv::GC_PR_FGD, result, cv::CMP_EQ);
// 生成输出图像
cv::Mat foreground(image.size(), CV_8UC3,
                   cv::Scalar(255, 255, 255));
image.copyTo(foreground, // 不复制背景像素
            result);
```

为了提取所有的前景像素，即数值等于 `cv::GC_PR_FGD` 或 `cv::GC_FGD` 的像素，仅仅需要核对第 1 个位中的值：

```
// 使用按位与 (bitwise-and) 核对第 1 个位
result = result & 1; // 前景时为 1
```

这是因为两个前景常量的值为 1 和 3，而另外两个的值为 0 和 2。在例子中，由于分割图像不包含 `cv::GC_FGD` 像素（只输入了 `cv::GC_BGD` 像素），可以得到相同的结果。最后，我们通过带掩码的复制运算，获取前景物体的图像（背景为白色）。

```
// 生成输出图像
cv::Mat foreground(image.size(), CV_8UC3,
                   cv::Scalar(255, 255, 255)); // 图像为全白
image.copyTo(foreground, result); // 不复制背景像素
```

结果如图 5.17 所示。

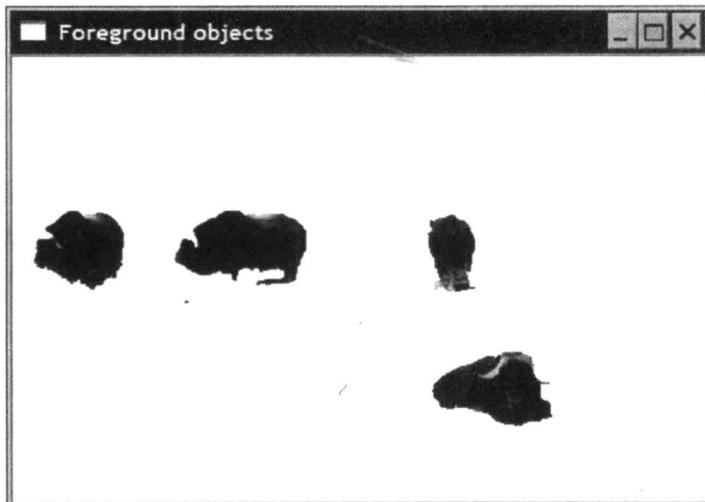


图 5.17

作用原理

在前述例子中，GrabCut 算法通过指定矩形包围盒（包含四个动物）来提取前景物体。同时，也可以将数值 `cv::GC_BGD` 和 `cv::GC_FGD` 赋予分割图像的某些特定像素，并且把这分割图像作为 `cv::grabCut` 函数的第二个参数。这时需要指定 `GC_INIT_WITH_MASK` 作为输入模式。这些标记也可以让用户以交互的方式进行输入。我们也可以组合使用这两个属于模式。

基于这些信息，GrabCut 通过下述步骤创建背景 / 前景分割。首先，前景标签 (`cv::GC_PR_FGD`) 被临时地赋予所有未标记的像素。基于当前的分类，算法将像素归类为色彩相似的聚类（背景和前景各有 K 个聚类）。下一步是通过引入背景与前景像素的边界进行分割。这个优化的过程尝试将标签相似的像素相连接，这利用了在强度相对已知的区域之间对边界像素的惩罚。这个最优化问题通过 GraphCut 算法得到高效解决，该算法将问题转化为对连通图进行切割 (Cut) 以构成最优配置。获取的分割结果产生新的像素标签。重复进行该聚类过程，会找到新的最优解，周而复始。因此 GrabCut 是一个迭代的过程，不断地优化结果。根据场景的复杂度，得到满意方案的迭代次数可多可少（对于简单场景，有时一次迭代便足够！）。

这解释了为何函数的最后一个参数需要指定迭代的次数。算法内部使用的两个模型作为函数参数传入（并传出），因此能够延续使用前一回合的模型，通过进行额外的迭代改进分割的结果。

参 考

GrabCut 算法的细节可以参阅 C.Rother, V.Kolmogorov 及 A.Blake 撰写的文章 “*GrabCut: Interactive Foreground Extraction using Iterated Graph Cuts*” [ACM Transactions on Graphics (SIGGRAPH), Vol.23, No.3, 2004.8]。

第 6 章

图像滤波

本章主要探讨：

- ◆ 使用低通滤波器；
- ◆ 使用中值滤波器；
- ◆ 使用方向滤波器检测边缘；
- ◆ 计算图像的拉普拉斯变换。

6.1 引言

滤波（Filtering）是信号处理及图像处理中的一个基本操作，旨在特定的应用程序中，选择性地提取图像中被认为传达重要信息的部分。滤波去除图像中的噪声，提取感兴趣的视觉特征，允许图像重采样，等等。它的根源来自于广义的信号与系统理论。我们不会在此讲述理论细节。然而，这章将介绍与滤波相关的一些重要概念，并展示滤波器如何用于图像处理程序。首先，让我们简要地解释一下频域分析的概念。

当观察一幅图像时，我们看到不同的灰度（或彩色值）在图像中的分布。图像间存在不同是因为它们有不同的灰度分布。但是，还存在另一种进行图像处理的方式——我们可以观察图像中存在的灰度变化。一些图像包含大面积近乎恒定的强度（例如，一个蓝色的天空），而在其他图像中，灰度强度变化迅速（例如，一个充满微小物体的热闹场景）。因此，观察图像中变化的频率构成了另一种描述图像的方式。这种观点被称为频域，而通过观察灰度分布来描述一幅图像被称为空间域。

频域分析按照高频到低频的次序，分解图像到频率内容。低频对应区域的图像强度变化缓慢，而高频是由快速变化的图像强度生成的。存在一些著名的变换，如傅里叶（Fourier）变换或余弦（Cosine）变换，它们可用来明确地显示图像的频谱。由于图像是二维的，它包含垂直频率（垂直方向的变化）和水平频率（水平方向的变化）。

在频域分析的框架下，滤波操作的作用是增强部分频段，同时限制（或衰减）其

他频段。低通 (Low-pass) 滤波器去除了图像中的高频成分，高通 (High-pass) 滤波器去除了低频成分。本章将展示一些频繁用于图像处理的滤波器，并解释它们对图像起到的效果。

6.2 使用低通滤波器

第一则秘诀介绍一些非常简单的低通滤波器。在前面的导论部分，我们了解到这类滤波器的目标是降低图像变化的幅度。一种简单的方案是将每个像素替换为相邻像素的平均值。通过这么做，快速的强度变化被转化为平缓的过渡。

实现方法

`cv::blur` 函数的作用正是将每个像素替换为相邻矩形内像素的平均值。这个低通滤波器的用法如下：

```
cv::blur(image,result,cv::Size(5,5));
```

这类滤波器又被称为箱式滤波器。这里我们使用 5×5 的滤波器已得到更显著的效果。当它应用于图 6.1 时，结果如图 6.2 所示。



图 6.1



图 6.2

在一些情况下，对于靠近的像素需要给予更多的的重要性。这可以通过计算加权平均数，即离得近的像素相比远处的像素拥有更多的权重。常用的加权方案基于高斯函数（Gaussian Function），这是一个“钟形的”函数。`cv::GaussianBlur` 函数便起到这个作用，它的用法如下：

```
cv::GaussianBlur(image, result, cv::Size(5,5), 1.5);
```

结果如图 6.3 所示。



图 6.3

作用原理

当滤波器的作用相当于将一个像素替换为相邻像素的加权总和时，我们称它是线性的。箱式滤波器便属于这一类，它将一个像素替换为相邻矩形内的像素和，并除以相邻像素的个数（以得到平均值）。这就像是将每个相邻像素乘以 1 后进行求和。滤波器的不同权重可以通过矩阵表示，矩阵的每一项都是对应位置像素的相乘因子。位于中心的元素对应的是滤波器当前正处理的像素。这样一个矩阵有时被称为核（Kernel）或掩码（Mask）。对于一个 3×3 的箱式滤波器，它对应的核是：

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

于是，应用一个线性滤波器意味着沿着图像的每个像素移动一个核，并且把每个对应的像素乘上关联的权重。在数学上，这个过程被称作卷积（Convolution）。

观察本秘诀生成的输出图像，不难看出低通滤波器的效果是对图像进行模糊或平滑。这并不令人惊讶，因为它减弱了物体边缘处可见的快速变化。

在高斯滤波器中，像素的权重与它离开中心像素点距离成正比。一维高斯函数有以下形式：

$$G(x) = A e^{-x^2/2\sigma^2}$$

归一化系数 A 使得不同权重之和为一。 σ （西格玛）数值控制高斯函数的高度。这个值越大，生成的函数越平坦。例如，对于间隔为 $[-4, \dots, 0, \dots, 4]$ 并且 $\sigma=0.5$ 的一维高斯滤波器，我们计算它的系数：

```
[0.0 0.0 0.00026 0.10645 0.78657 0.10645 0.00026 0.0 0.0]
```

当 $\sigma=1.5$ 时，这些系数则为

```
[0.00761 0.036075 0.10959 0.21345 0.26666 0.21345 0.10959 0.03608 0.00761]
```

这些值通过合适的 σ 值调用 `cv::getGaussianKernel` 函数即可得到：

```
cv::Mat gauss = cv::getGaussianKernel(9, sigma, CV_32F);
```

想要在一幅图像上应用二维高斯滤波器，可以先对图像的行应用一维高斯滤波器（这将影响垂直频率），接着对图像的列应用相同的一维滤波器（影响的是垂直频率）。

这利用的是高斯滤波器的可分离特性，即二维滤波器可以分解为两个一维滤波器）。函数 `cv::sepFilter2D` 可用于任意的可分离滤波器。也可以通过 `cv::filter2D` 函数来直接应用二维滤波。在 OpenCV 中，指定高斯滤波的方法是将系数的个数（第 3 个参数，必须是奇数）以及 σ 的值（第 4 个参数）提供给 `cv::GaussianBlur` 函数。如果指定滤波器的尺寸为 0，那么只需要设置 σ 的值，系数的合适数量将由 OpenCV 确定。反过来也可以，即提供尺寸的值，而设置 σ 为 0。然而，建议你同时设置这两个值以更好地控制滤波的效果。

扩展阅读

当图像大小改变时也将使用低通滤波器。假设你需要将尺寸减小为 2 的倍数，你也许认为可以简单地移除，但是这么做的效果将是很糟糕的。例如，原图中一条明显的边缘在缩小图中将显示为楼梯状，其他的锯齿失真也将出现于图像的曲线部分以及纹理丰富的部分。

这些不受欢迎的瑕疵由空间扭曲引起，这发生于将高频成分放入过小的图像时。事实上，小些的图像（即拥有更少像素点的图像）无法如同高分辨率图像一般表现出精美的纹理及尖锐的边缘（想象一下高清电视与传统电视的区别）。由于图像中的精致细节对应于高频分量，在将尺寸缩小前我们需要移除这些分量。在本秘诀中我们了解到低通滤波器可以达成该目的。也就是说，若想缩减一半的尺寸而不添加恼人的瑕疵，你必须先对原图应用低通滤波，然后隔行、隔列去除像素。这也正是 `cv::pyrDown` 函数的作用度：

```
cv::Mat reducedImage; // 包含缩小后的图像  
cv::pyrDown(image, reducedImage); // 将图像尺寸减半
```

此例使用 5×5 高斯滤波器对图像进行低通滤波。与之相反的 `cv::pyrUp` 函数也存在，用于将图像的尺寸放大一倍。当然，先缩小尺寸再放大并不能恢复原样。在缩小尺寸过程中丢失的信息是无法恢复的。这两个函数用于创建图像金字塔。这种数据结构由不同尺寸的图像叠加而成（通常每一层的尺寸都是上一层的一半），常常用于高效的图像分析。例如，如果想要检测图像中的物体，首先在金字塔顶层的小尺寸图像中进行检测，然后可以定位感兴趣的物体，并在包含高分辨图像的低层金字塔中进行更精确的搜索。

存在一个更通用的 `cv::resize` 函数允许你指定目标图像的尺寸。尺寸既可以小于原图，也可以大于原图。

```
cv::Mat resizedImage; // 包含改变尺寸后的图像  
cv::resize(image, resizedImage,  
cv::Size(image.cols/3, image.rows/3)); // 改变为 1/3 大小
```

在其他选项也可以通过设置防缩因子来确定尺寸，或是挑选重采样过程中的特定插值方式。

参 考

`cv::boxFilter` 函数使用由 1 组成的方形核对图像进行滤波，它类似于均值滤波器，除了不需要除以系数的个数。

第 2 章中“遍历图像和邻域操作”的“扩展阅读”部分介绍了 `cv::filter2D` 函数，它可以使用户自定义的核对图像进行线性滤波。

6.3 使用中值滤波器

本章的第一则秘诀介绍的是线性滤波器的概念。此外也存在非线性滤波器，它们有助于图像处理。本则秘诀介绍的中值滤波器便是其中一种。

由于中值滤波器对于去除椒盐 (Salt-and-Pepper) 噪点尤其有用，我们将使用第 2 章第一则秘诀中创建的图像（图 6.4）。



图 6.4

实现方法

调用中值滤波函数的方式与其他滤波很相似：

```
cv::medianBlur(image, result, 5);
```

生成的图像如图 6.5 所示。



图 6.5

作用原理

由于中值滤波器是非线性的，它无法表示为一个核矩阵。然而，它也对一个像素的相邻区域进行操作以确定输出像素的值。该像素及它的相邻区域组成一组数组，同时如名字所示，中值滤波器仅仅计算这组数的中值，并用中值替换当前的像素值。

这解释了为何该滤波器在去除椒盐噪声方面是如此的高效。事实上，当一个例外的黑色或白色像素出现在一个给定的相邻区域时，它从不会被选为中值（它或为极大值，或为极小值），因此它总替换为某个相邻像素的值。与之相反，一个简单的均值滤波器将会极大地受到这类噪声的影响，这可以从图 6.6 所示的均值滤波版本图像看出。



图 6.6

很明显，噪点像素使得相邻像素的平均值发生变化。即便已经被均值滤波器进行模糊，结果中的噪点依然是可见的。

中值滤波器同时还有保留边缘锐利度的优点。然而，它也会去除相同区域中的纹理（如背景中的树木）。

6.4 使用方向滤波器检测边缘

本章的第一则秘诀介绍了基于核矩阵的线性滤波，这些滤波器通过移除或减弱高频分量对图像进行模糊。在本则秘诀中，我们将进行相反的变换，即强调图像中的高频分量。我们将使用高通滤波器进行边缘检测。

实现方法

这里将使用的滤波器被称为 Sobel 滤波器。它具有方向性，根据使用的核的不同它只改变图像的水平或垂直频率。OpenCV 拥有一个对图像进行 Sobel 运算的函数。水平滤波器的使用方式如下：

```
cv::Sobel(image, sobelX, CV_8U, 1, 0, 3, 0.4, 128);
```

垂直滤波器的使用方式也类似：

```
cv::Sobel(image, sobelY, CV_8U, 0, 1, 3, 0.4, 128);
```

这个函数输入多个整数参数，我们将在稍后解释它们。这里先提一下，它们能够生成 8 位图像 (CV_8U) 来表示结果。

水平 Sobel 运算的结果如图 6.7 所示。

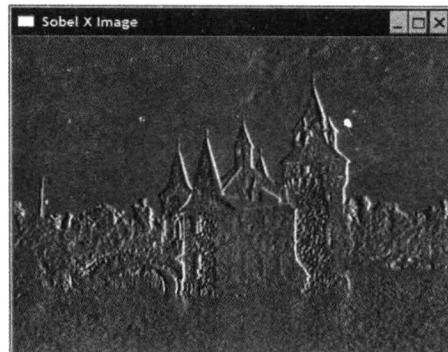


图 6.7

在这种表示方法中，零值对应的灰度值为 128，负数被表示为较暗的像素，而整数被表示为较亮的像素。垂直 Sobel 图像如图 6.8 所示。

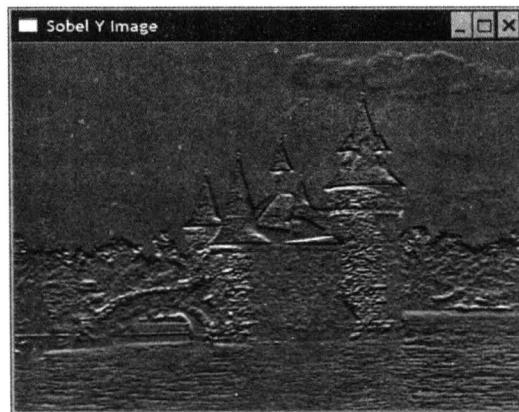


图 6.8

如果你熟悉相片编辑软件，那么先前的图像也许令你回想起浮雕特效，事实上这正是基于方向性滤波器。由于这个核包含正数和负数，Sobel 滤波器的结果通过保存在 16 位的有符号整数图像中 (CV_16S)。两个结果（垂直的和水平的）结合将得到 Sobel 滤波器的范式 (norm)。

```
// 计算 Sobel 范式
cv::Sobel(image, sobelX, CV_16S, 1, 0);
cv::Sobel(image, sobelY, CV_16S, 0, 1);
cv::Mat sobel;
// 计算 L1 范式
sobel= abs(sobelX)+abs(sobelY);
```

通过 convertTo 函数可选的缩放参数，Sobel 距离可以方便地显示在一幅图像中，零值对应于白色，而较高的值被赋予暗灰色：

```
// 搜索 Sobel 极大值
double sobmin, sobmax;
cv::minMaxLoc(sobel, &sobmin, &sobmax);
// 变换为 8 位图像
// sobelImage= -alpha*sobel+255
cv::Mat sobelImage;
sobel.convertTo(sobelImage, CV_8U, -255./sobmax, 255);
```

结果如图 6.9 所示。



图 6.9

观察这幅图像，可以了解到为何这类算子被称为边缘检测器。之后，通过阈值化可以得到二值图像，表示的是图像的轮廓。下述代码用于创建之后的图像（图 6.10）：

```
cv::threshold(sobelImage, sobelThresholded,  
             threshold, 255, cv::THRESH_BINARY);
```

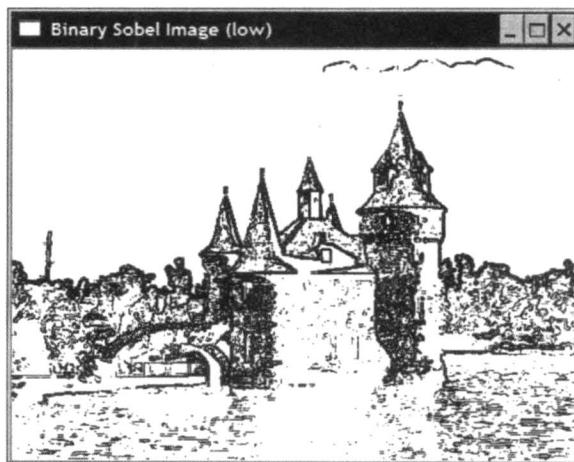


图 6.10

作用原理

Sobel 算子是一种经典的边缘检测线性滤波器，它基于一个简单的 3×3 核：

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

如果我们将图像视作二维函数，Sobel 算子可被认为是图像在垂直和水平方向变化的测量。这种测量在数学中被称为梯度，它被定义为由函数在两个正交方向上的一阶导数组成的二维向量：

$$\text{grad}(I) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right]^T$$

因此，Sobel 算子通过在水平和垂直方向下进行像素差分给出图像梯度的近似。它在感兴趣像素的小窗口内运算，以减少噪点的影响。`cv::Sobel` 函数使用 Sobel 核计算图像卷积的结果。它的完整使用说明如下：

```
cv::Sobel(image,           // 输入
          sobel,        // 输出
          image_depth,  // 图像类型
          xorder, yorder, // 核的阶数
          kernel_size, // 方形核的大小
          alpha, beta); // 缩放值及偏移量
```

你可以选择输出图像的类型为无符号字符、有符号字符或是浮点数。当然，如果结果位于像素值域之外，将进行饱和截断，这也是最后两个参数起作用的地方。在结果被保存前，可以乘以 `alpha` 变量，再加上 `beta` 偏移量。这是我们在前一部分生成的图像，Sobel 值为 0 被表示为灰度 128。每个 Sobel 值对应着一个方向上的导数。因此，两个参数被用于指定卷积核， x 及 y 方向上的导数的阶数。

例如，水平 Sobel 核的使用方式是 x 方向阶数为 1 而 y 方向阶数为 0，垂直 Sobel 核的则是分别为 0 和 1。其他的组合也是有效的，但是这两种使用最频繁（二阶导数将在下则秘诀中讨论）。最后，也可以使用尺寸大于 3×3 的核。1、3、5 及 7 都是可选的尺寸值。尺寸为 1 的核对应的是一维的 Sobel 滤波器 (1×3 或 3×1)。

由于梯度是一个二维向量，它拥有距离和方向。梯度向量的距离给出变化的幅度，通常使用的是欧拉距离（也被称为 L2 距离）：

$$|\text{grad}(I)| = \sqrt{\left(\frac{\partial I}{\partial x} \right)^2 + \left(\frac{\partial I}{\partial y} \right)^2}$$

然而，在图像处理中，我们通常计算的是绝对值之和，也就是 L1 距离，它的结果与 L2 距离相近但是复杂度要低得多。我们在这则秘诀里这么做：

```
// 计算 L1 范式  
sobel= abs(sobelX)+abs(sobelY);
```

梯度向量总是指向变化最剧烈的方向。在图像中，这意味着梯度向量总是与边缘正交，从暗处指向亮处。梯度的方向为：

$$\angle \text{grad}(I) = \alpha \tan\left(-\frac{\partial I}{\partial y} / \frac{\partial I}{\partial x}\right)$$

边缘检测中，通常只计算距离。但是如果既需要距离又需要方向，那么可以使用下述 OpenCV 方向：

```
//Sobel 必须以浮点数格式进行计算  
cv::Sobel(image,sobelX,CV_32F,1,0);  
cv::Sobel(image,sobelY,CV_32F,0,1);  
// 计算 L2 范式及梯度方向  
cv::Mat norm,dir;  
cv::cartToPolar(sobelX,sobelY,norm,dir);
```

默认情况下，方向的单位是弧度。设置一个额外的参数为 `true` 即可得到角度值。

对梯度的大小进行阈值化后可以得到二值的边缘图。选择合适的阈值很有难度。如果太低，那么会保留过多（厚的）边，同时太高的值会导致破碎的边。图 6.11 使用的是一个较高的阈值，与之前得到的二值边缘图进行比较可以看出这种区别。

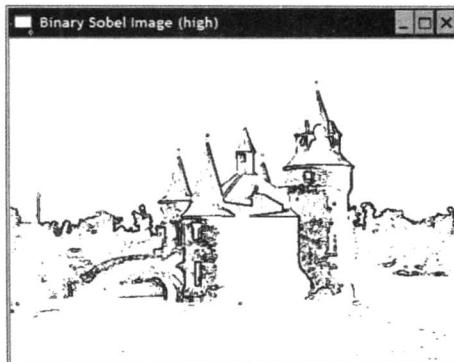


图 6.11

一种可选的途径是使用磁滞阈值化的概念。这将在下章介绍 Canny 算子的时候解释。

扩展阅读

其中的梯度算子也存在。例如，Prewitt 算子的核定义如下：

$$\begin{array}{ccc} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{array} \quad \begin{array}{ccc} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{array}$$

Roberts 算子则是基于这些简单的 2×2 核：

$$\begin{array}{cc} 1 & 0 \\ 0 & -1 \end{array} \quad \begin{array}{cc} 0 & 1 \\ -1 & 0 \end{array}$$

Scharr 算子在需要对梯度朝向有精确估计时效果很好：

$$\begin{array}{ccc} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{array} \quad \begin{array}{ccc} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 0 & 3 \end{array}$$

注意：`cv::Sobel` 函数配上参数 `CV_SCHARR` 也可以使用 Scharr 核：

```
cv::Sobel(image, sobelX, CV_16S, 1, 0, CV_SCHARR);
```

或者，直接调用 `cv::Scharr` 函数：

```
cv::Scharr(image, scharrX, CV_16S, 1, 0, 3);
```

这些方向性滤波器都尝试估计图像函数的一阶导数。因此，在滤波器方向上变化剧烈的区域对应的是高值，而平坦区域生成的是低值。这便是计算图像导数的滤波器是高通滤波器的原因。

参 考

第 7 章中的“使用 Canny 算子检测轮廓”使用两个阈值来提取一幅二值边缘图。

6.5 计算图像的拉普拉斯变换

Laplacian（拉普拉斯）是另一种基于图像导数的高通线性滤波器，它计算二阶导数以衡量图像的弯曲度。

实现方法

OpenCV 函数 `cv::Laplacian` 计算一幅图像的 Laplacian，它与 `cv::Sobel` 函数很类似。事实上它使用同一个基础函数 `cv::getDerivKernels` 来获取核矩阵，唯一的差别是不存在指定导数阶数的参数，因为它们都是二阶导数。

对于这个算子，我们将创建一个简单的类来封装一些有用的操作。基本方法：

```
class LaplacianZC{
private:
    // 原图
    cv::Mat img;
    // 包含 Laplacian 的 32 位浮点图像
    cv::Mat laplace;
    //laplacian 卷积核的大小
    int aperture;
public:
    LaplacianZC():aperture(3){}
    // 设置卷积核的大小
    void setAperture(int a){
        aperture= a;
    }
    // 计算浮点数 Laplacian
    cv::Mat computeLaplacian(const cv::Mat& image){
        // 计算 Laplacian
        cv::Laplacian(image,laplace,CV_32F,aperture);
        // 保留图像的局部备份（用于零点交叉）
        img= image.clone();
        return laplace;
    }
}
```

Laplacian 的计算结果保存在浮点数图像中。为了得到结果，我们进行与之前秘诀相同的重缩放。这个重缩放基于拉普拉斯最大绝对值，其中 0 对应灰度 128。我们的

类的方法允许得到该图像：

```
// 返回 8 位图像存储的 Laplacian 结果  
// 零点交叉于灰度值 128  
// 如果没有指定 scale 参数，那么最大值将缩放至强度 255  
// 你必须在调用它之前调用 computeLaplacian  
cv::Mat getLaplacianImage(double scale=-1.0){  
    if(scale<0){  
        double lapmin,lapmax;  
        cv::minMaxLoc(laplace,&lapmin,&lapmax);  
        scale= 127/std::max(-lapmin,lapmax);  
    }  
    cv::Mat laplaceImage;  
    laplace.convertTo(laplaceImage,CV_8U,scale,128);  
    return laplaceImage;  
}
```

使用该类，可以这么计算图像的 7×7 拉普拉斯变换：

```
// 使用 LaplacianZC 类进行计算  
LaplacianZC laplacian;  
laplacian.setAperture(7);  
cv::Mat flap=laplacian.computeLaplacian(image);  
laplace=laplacian.getLaplacianImage();
```

结果如图 6.12 所示。



图 6.12

作用原理

2D 函数的拉普拉斯变换定义为它的二阶导数之和：

$$\text{laplace}(I) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

使用最简单的形式，它可以使用下述 3×3 核近似：

0	1	0
1	-4	1
0	1	0

与 Sobel 算子相同，也能够使用更大的核计算 Laplacian，同时由于这个运算对于噪点更敏感，我们倾向于这么做（除非计算效率更重要）。需要注意 Laplacian 核的总数为 0，这保证了强度不变区域的 Laplacian 为 0。事实上，由于 Laplacian 度量的是图像函数的曲率，它在平坦区域应该等于 0。

乍一看，拉普拉斯算子的效果可能很难解释。从核的定义来看，很明显，任何孤立的像素值（它与相邻像素值截然不同）将被算子放大。这源于算子对噪点的高灵敏度。但是拉普拉斯算子值在图像边缘处的表现更有趣。边缘的存在是图像中不同灰度区域间快速过渡的结果。沿着图像在一条边上的变化（例如，从暗处到亮处），可以观察到灰度的提升意味着从正曲率（强度值开始上升）到负曲率（当强度即将达到最高值）的渐变。因此，正、负拉普拉斯算子值（或导数）之间的过渡是存在边缘的指示器。另一种表达这个事实的方法是说，边缘位于 Laplacian 函数的零交点。我们将通过查看测试图像小窗口中的拉普拉斯算子值来阐述这个观点。我们选择的部分对应于由城堡塔楼的屋顶底部带来的边缘。一个白色的盒子被绘制于图 6.13，以显示感兴趣区域的确切位置。



图 6.13

现在，观察窗口中的 Laplacian 值（ 7×7 核），我们得到：

25	-78	-140	-115	-59	-23	-5	-3	-6	-2	1	2
46	-7	-127	-186	-148	-73	-21	-11	-12	-4	0	-1
-40	16	-46	-164	-213	-165	-84	-33	-10	4	2	-7
-6	84	105	-7	-121	-185	-158	-88	-29	0	2	-8
135	202	284	296	199	20	-109	-106	-47	-10	3	0
124	193	316	438	442	251	16	-69	-41	-9	8	8
39	111	216	287	254	123	-19	-69	-42	-12	6	8
38	110	180	85	-131	-217	-161	-90	-37	-7	0	-3
36	121	210	31	-299	-355	-183	-61	-12	5	0	-10
1	95	210	43	-271	-293	-111	-17	0	6	1	-4
-8	102	208	48	-227	-228	-62	4	3	2	2	3
19	167	247	51	-225	-220	-57	6	3	0	0	0

仔细地沿着 Laplacian 零交点（位于正负不同像素之间），便能得到一条对应于图像窗口中可见边缘的曲线。我们沿着零交点绘制点缀的直线，对应于可见的塔楼中的边缘。这意味着，原则上，你可以在亚像素（Sub-pixel）精度检测图像边缘。

跟随 Laplacian 图像中零交叉曲线是一件容易出错的事情。然而，一个简化的算法被用于检测近似的零交叉位置，过程如下。扫描 Laplacian 图像，比较当前像素与左侧像素。如果两者的符号不同，那么说明当前像素存在零交叉。如果没有，在正上方位置重复相同测试。这个算法使用下述方法实现，能够生产零交叉的二值图像：

```
// 得到零点交叉的二值图像
// 如果相邻像素的乘积小于 threshold
// 那么零点交叉将被忽略
cv::Mat getZeroCrossings(float threshold=1.0) {
    // 创建迭代器
    cv::Mat<float>::const_iterator it=
        laplace.begin<float>()+laplace.step1();
    cv::Mat<float>::const_iterator itend=
        laplace.end<float>();
    cv::Mat<float>::const_iterator itup=
        laplace.begin<float>();
    // 初始化为白色的二值图像
    cv::Mat binary(laplace.size(), CV_8U, cv::Scalar(255));
    cv::Mat<uchar>::iterator itout=
        binary.begin<uchar>()+binary.step1();
    // 对输入阈值取反
    threshold*= -1.0;
    for(;it!= itend;++it,++itup,++itout)
        // 如果相邻像素的乘积为负数，那么符号发生改变
```

```

    if(*it** (it-1)<threshold)
        *itout= 0;// 水平方向零点交叉
    else if(*it**itup<threshold)
        *itout= 0;// 垂直方向零点交叉
    }
    return binary;
}

```

使用额外的阈值化来确保当前的 Laplacian 值可以被认为是一条边。得到的结果为下述二值图像，如图 6.14 所示。

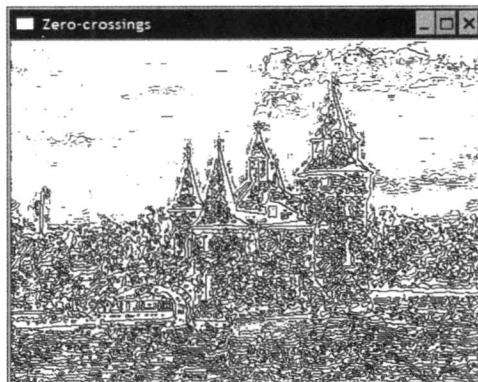


图 6.14

你知道，Laplacian 的零交点检测到所有边缘，强边与弱边之间不做区分。我们提到过 Laplacian 对于噪点很敏感，这两个原因解释了为何该算子会检测到这么多的边缘。

扩展阅读

让一幅图像减去它的 Laplacian 可以增强对比度。这正是我们在第 2 章的“遍历图像和邻域操作”中介绍下述卷积核时所做的：

$$\begin{array}{ccc} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 1 \end{array}$$

它等于 1 减去 Laplacian 核（即原图减去它的 Laplacian）。

参 考

第 8 章的“检测尺度不变的 SURF 特征”在检测尺度不变的特征时使用了 Laplacian。

第 7 章

提取直线、轮廓及连通区域

本章主要探讨：

- ◆ 使用 Canny 算子检测轮廓；
- ◆ 使用霍夫变换检测直线；
- ◆ 用直线拟合一组点；
- ◆ 提取连通区域的轮廓；
- ◆ 计算连通区域的形状描述符。

7.1 引言

为了对图像实施基于内容的分析，需要从像素之中提取有意义的特征。轮廓、直线、像素块等都是定义图像内容的基本元素。本章将介绍如何提取这些重要的图像特征。

7.2 使用Canny算子检测轮廓

在前一章，我们学习到如何检测图像边缘。尤其是，我们展示了对梯度大小进行阈值化以得到二值的边缘图像。边缘包含着重要的视觉信息，因为它们描绘出图像元素的轮廓。因为这个原因，它们可以用于物体检测。然而，简单的二值边缘图像有两大缺陷。第一点，检测到的边缘过粗，这意味着难以实现物体的精确定位。第二点，也是最重要的一点，难以找到这样一个阈值，既能足够低以检测到所有重要边缘，同时又不至于包含过多次要的边缘。这个需要权衡的问题正是 Canny 算法尝试解决的。

实现方法

OpenCV 中实现 Canny 算法的函数是 `cv::Canny`。正如我们将看到的，该算法需

要指定两个阈值。函数的调用方法如下：

```
// 应用 Canny 算法  
cv::Mat contours;  
cv::Canny(image, // 灰度图  
          contours, // 输出轮廓  
          125, // 低阈值  
          350); // 高阈值
```

当使用在图 7.1 所示的图像上时，结果如图 7.2 所示。



图 7.1

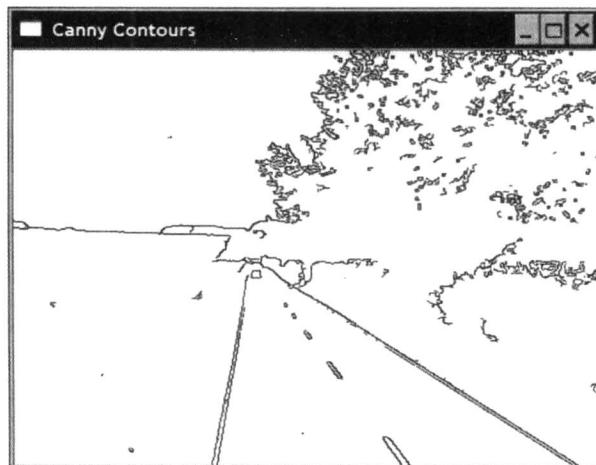


图 7.2

注意，为了得到先前截屏中所示的图像，我们必须反转黑白值，这是因为正常情况下轮廓用非零像素表示。反转后的图像更适合打印在纸上，它的生成方式如下：

```
cv::Mat contoursInv; // 反转后的图像
cv::threshold(contours, contoursInv,
              128, // 低于该值的像素
              255, // 将变成 255
              cv::THRESH_BINARY_INV);
```

作用原理

Canny 算子通常基于 Sobel 算子，尽管也可以使用其他梯度算子。核心思想是使用两个不同阈值以确定哪些点属于轮廓：一个低值和一个高值。

挑选的低值应该包括所有被认为是属于明显图像轮廓的边缘像素。例如，使用之前例子中指定的低阈值，并且将它应用于 Sobel 算子的结果，那么可以得到图 7.3 所示的边缘图像。



图 7.3

如图所示，描绘道路的边缘非常明显。然而，由于使用了宽松的阈值，不必要的边缘也被检测到。第二个阈值的角色正是定义属于所有重要轮廓的边缘，它应该排除所有异常值。例如，对于本例中高阈值的 Sobel 边缘图像如图 7.4 所示。



图 7.4

现在我们的图像包含分离的边缘，但是它们当然属于屏幕中的重要轮廓。Canny 算法组合这两幅边缘图以生成一幅“最优的”轮廓图。如果存在连续的边缘路径，将低阈值图像中的边缘点与高阈值图像中的边相连接，那么就保留这个低阈值图像中的边缘点。因此，高阈值图像中所有的边缘点都保留下，同时移除低阈值图像中所有孤立的点。只要指定合适的阈值，那么这个方案都能生成高质量的轮廓。这种使用双阈值以得到二值图像的策略被称为磁滞阈值化，可用于任何通过阈值化获取二值图像的情况。然而，这样做的代价是更高的计算复杂度。

此外，Canny 算法使用额外策略以提升图像质量。在使用磁滞阈值化前，所有在梯度大小并非最大值的边缘点都被移除。梯度的朝向总是与边缘垂直，因此该方向的局部梯度最大值对应的是轮廓强度最大的点。这解释了为何 Canny 算子能得到薄的边缘。

参 考

推荐阅读 J.Canny 撰写经典论文 “*A Computational Approach to Edge Detection*” (IEEE Transactions on Pattern Analysis and Image Understanding, Vol.18, No.6, 1986)。

7.3 使用霍夫变换检测直线

人类世界中充满着平面的及线性的结构，于是直线在图像中频繁可见。这些

富有意义的特征在物体识别和图像理解领域扮演着重要的角色。霍夫变换（Hough Transform）是检测直线的经典算法。它最初只用于检测直线，被扩展后能够检测其他的简单结构。

准备工作

在霍夫变换中，直线用下述方程表示：

$$\rho = x \cos\theta + y \sin\theta$$

参数 ρ 是直线到图像原点（左上角）的距离， θ 则是与直线垂直的角度。在这种表达方式下，图像中可见直线的角度 θ 位于弧度 0 到 π 之间，而半角 ρ 的最大值等于图像的对角线长度。例如，思考图 7.5 中的直线。

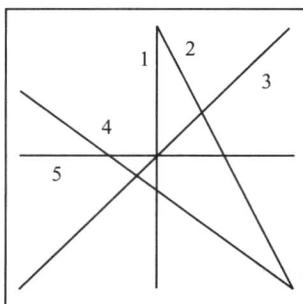


图 7.5

类似直线 1 的垂线的角度 θ 等于 0，而水平线（如直线 5）的角度 θ 等于 $\pi/2$ 。同时，直线 3 的角度 θ 等于 $\pi/4$ ，而直线 4 的角度大约是 0.7π 。为了能够在 θ 为区间 $[0, \pi]$ 之间表现所有可能的直线，半径值可取负数。这正是直线 2 的情况，它的 θ 值等于 0.8π ，而 ρ 为负值。

实现方法

OpenCV 提供两种霍夫变换的实现，基础版本是 `cv::HoughLines`。它的输入为一幅包含一组点（表示为非零像素）的二值图像，其中的一些排列后形成直线。通常这是一幅边缘图像，比如来自 Canny 算子。`cv::HoughLines` 函数的输出是 `cv::Vec2f` 向量，每个元素都是一对代表检测到的直线的浮点数 (ρ, θ)。在下例中，我们首先应用 Canny 算子获取图像轮廓，然后基于霍夫变换检测直线：

```
// 应用 Canny 算法
cv::Mat contours;
cv::Canny(image, contours, 125, 350);
//Hough 变换检测直线
std::vector<cv::Vec2f> lines;
cv::HoughLines(test, lines,
    1, PI/180,          // 步进尺寸
    80);                // 最小投票数
```

第三个和第四个参数对应的是直线搜索时的步进尺寸。在我们的例子中，该函数以 1 为间隔搜索所有可能的半径，同时以 $\pi/180$ 为间隔搜索所有可能的角度。最后一个参数的角色将在稍后解释。当前参数选择下，路面图像共检测到 15 条线。为了可视化检测结果，我们将这些直线绘制在原始图像上。然而，需要强调算法检测的是直线，而非线段，我们无法得到线的端点。因此，我们绘制的直线将穿过整幅图像。对于几乎垂直的线，我们计算它与图像水平界线的交集（即第一行与最后一行）并在这两个点之间绘制直线。对于近似水平的直线，我们用同样的处理方法，只是换成第一列与最后一列。使用 `cv::line` 函数可以绘制直线，该函数对于位于图像以外坐标点也能良好运作。因此，必须要检测交点是否落在图像内。通过遍历向量，我们绘制每条线：

```
std::vector<cv::Vec2f>::const_iterator it= lines.begin();
while(it!=lines.end()){
    float rho= (*it)[0];           // 第一个参数为距离 rho
    float theta= (*it)[1];         // 第二个参数为角度 theta
    if(theta<PI/4.
        ||theta>3.*PI/4.){//~ 垂直线
        // 线与第一行的交点
        cv::Point pt1(rho/cos(theta),0);
        // 线与最后一行的交点
        cv::Point pt2((rho-result.rows*sin(theta))/
                      cos(theta),result.rows);
        // 绘制白线
        cv::line(image,pt1,pt2,cv::Scalar(255),1);
    }else{//~ 水平线
        // 线与第一列的交点
        cv::Point pt1(0,rho/sin(theta));
        // 线与最后一列的交点
        cv::Point pt2(result.cols,
```

```

        (rho-result.cols*cos(theta))/sin(theta));
    // 绘制白线
    cv::line(image,pt1,pt2, cv::Scalar(255),1);
}
++it;
}

```

结果如图 7.6 所示。

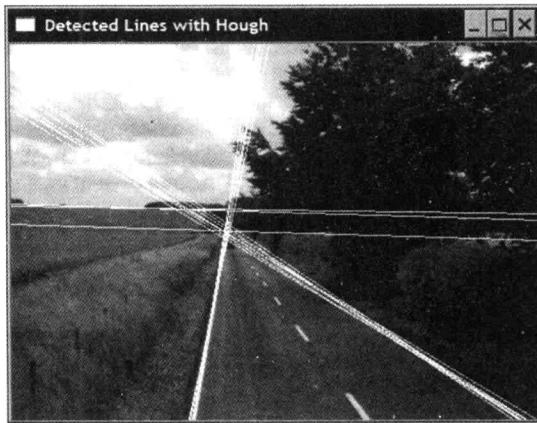


图 7.6

霍夫变换仅仅查找边缘点的一种排列方式，由于意外的像素排列或是多条线穿过同一组像素，很有可能带来错误的检测。

为了克服这些难题，同时检测到带端点的线段，人们提出了改进后的算法，即概率霍夫变换（Probabilistic Hough Transform），对应的函数时 `cv::HoughLinesP`。我们将它封装在 `LineFinder` 类中：

```

class LineFinder{
private:
    // 原图
    cv::Mat img;
    // 向量中包含检测到的直线的端点
    std::vector<cv::Vec4i> lines;
    // 累加器的分辨率
    double deltaRho;
    double deltaTheta;
    // 直线被接受时所需的最小投票数
    int minVote;
}

```

```

// 直线的最小长度
double minLength;
// 沿着直线方向的最大缺口
double maxGap;
public:
    // 默认的累加器的分辨率为单个像素及 1°
    // 不设置缺口及最小长度的值
    LineFinder():deltaRho(1),deltaTheta(PI/180),
        minVote(10),minLength(0.),maxGap(0.){}

```

相应的设置方法为

```

// 设置累加器的分辨率
void setAccResolution(double dRho,double dTheta){
    deltaRho= dRho;
    deltaTheta= dTheta;
}
// 设置最小投票数
void setMinVote(int minv){
    minVote= minv;
}
// 设置缺口及最小长度
void setLineLengthAndGap(double length,double gap){
    minLength= length;
    maxGap= gap;
}

```

于是，可以这么调用霍夫线段检测：

```

// 使用概率霍夫变换
std::vector<cv::Vec4i>findLines(cv::Mat& binary) {
    lines.clear();
    cv::HoughLinesP(binary,lines,
                    deltaRho,deltaTheta,minVote,
                    minLength,maxGap);
    return lines;
}

```

该方法返回 `cv::Vec4i` 类型的向量，每个都包含线段的起点和终点坐标，之后可以这么进行绘制：

```
// 绘制检测到的直线
void drawDetectedLines(cv::Mat &image,
    cv::Scalar color=cv::Scalar(255,255,255)){
    // 画线
    std::vector<cv::Vec4i>::const_iterator it2=
        lines.begin();
    while(it2!=lines.end()){
        cv::Point pt1((*it2)[0],(*it2)[1]);
        cv::Point pt2((*it2)[2],(*it2)[3]);
        cv::line(image,pt1,pt2,color);
        ++it2;
    }
}
```

现在，使用同一个输入图形，直线可以按如下顺序进行检测：

```
// 创建 LineFinder 实例
LineFinder finder;
// 设置概率 Hough 参数
finder.setLineLengthAndGap(100,20);
finder.setMinVote(80);
// 检测并绘制直线
std::vector<cv::Vec4i> lines= finder.findLines(contours);
finder.drawDetectedLines(image);
cv::namedWindow("Detected Lines with HoughP");
cv::imshow("Detected Lines with HoughP",image);
```

得到的结果如图 7.7 所示。

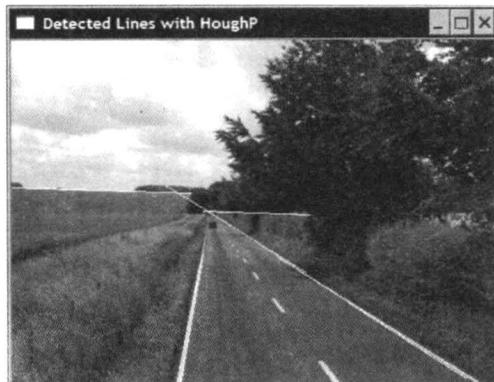


图 7.7

作用原理

霍夫变换的目的是找到二值图像中经过足够多数量的点的所有直线，它分析每个单独的像素点，并识别出所有可能经过它的直线。当同一条直线穿过许多点，便意味着这条线的存在足够明显。

霍夫变换使用二维的累加器以统计特定的直线被识别了多少次。该累加器的尺寸由指定的 (ρ, θ) 参数的步进尺寸定义（正如之前所提到的）。为了描绘该变换的作用，我们创建一个 180×200 的矩阵（对应于 θ 的步进为 $\pi/180$ ，对应于 ρ 为 1）：

```
// 创建霍夫累加器
// 此处为 uchar 图像；实践中应该使用 int 类型
cv::Mat acc(200,180,CV_8U,cv::Scalar(0));
```

该累加器是不同 (ρ, θ) 值的映射。因此，矩阵中的每一项都对应着一条特定的直线。我们考虑一个点，比如说，位于坐标 $(50, 30)$ ，通过遍历所有可能的 θ 角度（步进为 $\pi/180$ ）我们能识别出所有穿过它的直线，并且计算对应的 ρ 值：

```
// 选择点
int x=50,y=30;
// 遍历所有角度
for(int i=0;i<180;i++) {
    double theta= i*PI/180.;
    // 寻找对应的 rho 值
    double rho= x*cos(theta)+y*sin(theta);
    // j 对应的 rho 从 -100 到 100
    int j= static_cast<int>(rho+100.5);
    std::cout<<i<<","<<j<<std::endl;
    // 增加累加器
    acc.at<uchar>(j,i)++;
}
```

累加器中对应于 (ρ, θ) 对的项会自增，表示所有这些穿过该点的线（或者说，每个点为一组可能的候选直线进行投票）。如果我们在图像中显示累加器的值（乘以 100 以使得计数为 1 时也可见），那么可以得到图 7.8。

这条曲线代表着所有经过指定点的线。如果我们对于点 $(30, 10)$ 重复同样的试验，我们得到图 7.9 所示的累加器。

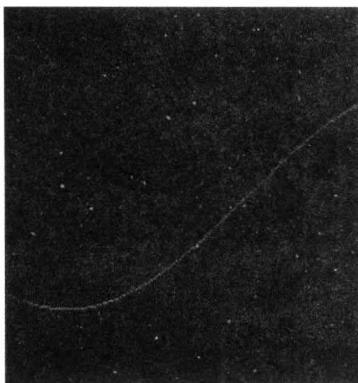


图 7.8

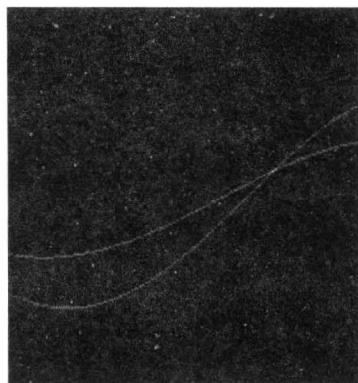


图 7.9

如图所示，两条生成的曲线相交于一点，这对应于穿过两个点的直线。累加器中对应的项得到两次投票，表示这条直线穿过两个点。如果我们将图像中所有的点进行同样的操作，那么沿着特定直线排列的点将使得累加器中相同的项增加许多次。最后，我们只需要识别累加器中的局部最大值，即得到大量投票的项，以检测图像中的直线（即点的排列）。`cv::HoughLines` 中最后一个参数对应的是能够被检测为直线所需要的最小投票数目。如果将该值降低到 60，即：

```
cv::HoughLines(test, lines, 1, PI/180, 60);
```

那么将会检测到更多的直线，如图 7.10 所示。

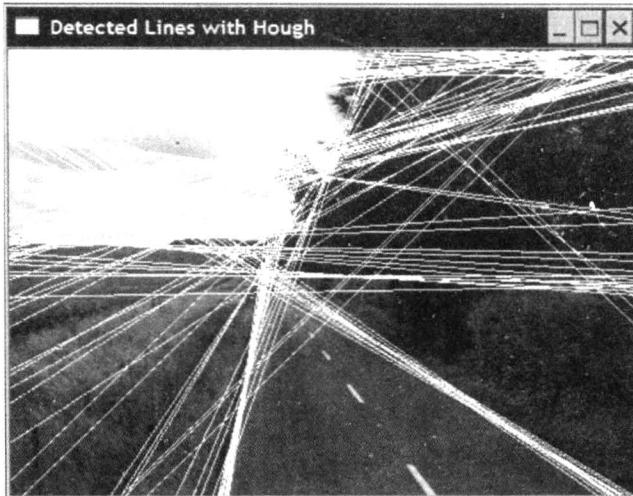


图 7.10

概率霍夫变换在原算法基础上增加了少许改动。首先，我们不在系统地逐行扫描图像，而是随机挑选像素点。一旦累加器中的某一项达到给定的最小值，那么扫描沿着对应直线的像素并移除所有经过的点（即便它们并未投过票）。这次扫描同时决定接受的线段的长度。为此，该算法定义两个额外参数：一个是可以接受的线段的最小长度，另一个是允许组成连续线段的最大像素间隔。额外的步骤增加了算法的复杂度，但是由于参与投票的点数量减少（直线扫描过程移除了部分点）得到部分的补偿。

扩展阅读

霍夫变换也可用于检测其他几何体。事实上，任何可用参数方程表示的几何体都可尝试用霍夫变换进行检测。

1. 检测圆

在检测圆的例子中，对应的参数方程为

$$r^2 = (x - x_0)^2 + (y - y_0)^2$$

该方程包括三个参数（圆的半径及圆心坐标），意味着需要三维的累加器。然而，通常而言霍夫变换随着维护的增加变得不那么可靠。在这个例子中，每个点都会使得累加器中大量的项被增加，因此，精确地定位局部峰值变得更加困难。对于解决这个问题我们有不同的策略。OpenCV 中实现的霍夫圆检测算法使用两个步骤。第一步，二维累加器用于找到可能的圆的位置。由于在圆周上的点的梯度应该指向半径的方向，因此，对于每个点，只有沿着梯度方向的项才得到增加（基于预定义的最大与最小半径值）。一旦找到可能的圆心（即获得预定义数量的投票），我们便在第二步构建一维的半径的直方图。这个直方图中的峰值对应的是检测到的圆的半径。

实现了该策略的 `cv::HoughCircles` 函数整合了 Canny 检测与霍夫变换。它的使用方法如下：

```
cv::GaussianBlur(image, image, cv::Size(5,5), 1.5);
std::vector<cv::Vec3f> circles;
cv::HoughCircles(image, circles, CV_HOUGH_GRADIENT,
    2,           // 累加器的分辨率 (图像的尺寸 /2)
    50,          // 两个圆之间的最小距离
    200,         // Canny 高阈值
    100,         // 最小投票数
    25,100);    // 极小极大半径
```

我们总是建议在调用 `cv::HoughCircles` 之前对图像进行平滑，这样可以减少可能引起误检测的图像噪声。检测的结果以 `cv::Vec3f` 向量的形式给出。头两个参数是圆心坐标，第 3 个参数是半径。在撰写本书时，参数 `CV_HOUGH_GRADIENT` 是唯一有效的选项，它对应的是两步检测法。第 4 个参数定义了累加器的分辨率。它是一个除法因子，例如，指定 2 将设置累加器大小为图像尺寸的一半。第 5 个参数是两个检测到的圆之间的最小距离。第 6 个参数对应于 Canny 边缘检测器的高阈值，低阈值被设置为它的一半。第 7 个参数是在第一步中，被检测到的圆心需要得到的最小投票数。最后两个参数是有效半径的最小最大值。由于涉及许多参数，该函数很难调节到理想值。

一旦获得检测到的圆的向量，可以通过遍历该向量并调用 `cv::circle` 函数进行绘制：

```
std::vector<cv::Vec3f>::  
    const_iterator itc= circles.begin();  
while(itc!=circles.end()) {  
    cv::circle(image,  
    cv::Point((*itc)[0], (*itc)[1]), // 圆心  
    (*itc)[2], // 半径  
    cv::Scalar(255), // 颜色  
    2); // 厚度  
    ++itc;  
}
```

图 7.11 是在选定参数下得到的结果。

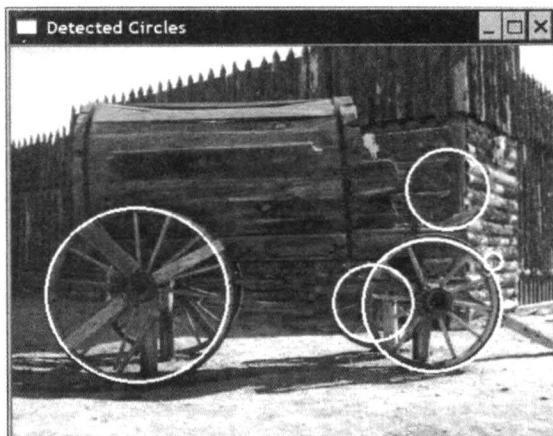


图 7.11

2. 广义霍夫变换

对于一些形状，很难找到简洁的参数化表现方式，如三角形、八角形、多边形、物体轮廓等。然而，还是可能使用霍夫变换定位这些形状，用到的原理是相同的。创建一个二维的累加器，用于表示所有可能存在目标形状的位置。因此，必须定义一个参考点，图像上的每个特征点都对可能的参考点坐标进行投票。由于一个点可能位于形状轮廓内任意位置，所有可能的参考位置将在累加器中描绘出一个形状，它将是目标形状的镜像。即，属于同一个形状的点将在累加器中生成峰值，并且该位置对应于形状所在的位置。

图 7.12 中的目标形状是三角形（位于右边），参考点定位于左侧底部角落。在累加器中一个特征点将增加所有绘制位置的项，因为它们都对应于三角形中参考点可能穿过该特征点的位置。

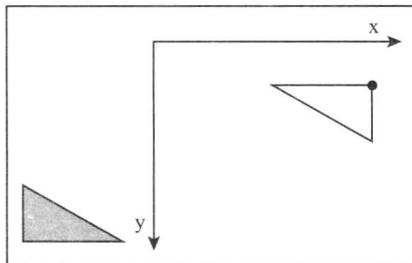


图 7.12

这个方法通常被称为广义霍夫变换。显而易见，它没有考虑缩放与旋转的影响，这需要更高维度空间中的搜索。

参 考

由 C.Galambos, J.Kittler, and J.Matas 撰写的 “*Gradient-based Progressive Probabilistic Hough Transform*” (*IEE Vision Image and Signal Processing*, Vol.148, No.3, pp.158-165, 2002) 是霍夫变换参考文章中的一篇，并且描述了 OpenCV 中实现的概率算法。

7.4 用直线拟合一组点

在一些应用中，不仅需要检测直线，还要获取直线位置和方向的精确估计。本秘诀将展示如何找到最贴合一组点的直线。

实现方法

首先要做的是识别出可能排列成直线的点，我们将使用前一则秘诀中检测到的一条直线。假设 `cv::HoughLinesP` 检测的直线被保存在名为 `lines` 的 `std::vector` 对象中。为了提取可能属于其中第一条线的点，我们在黑色图像上绘制白线，并且将它与用于检测我们直线的轮廓的 Canny 图像相交。做法如下：

```
int n=0; // 选择 line 0
// 黑色图像
cv::Mat oneline(contours.size(), CV_8U, cv::Scalar(0));
// 白色直线
cv::line(oneline,
         cv::Point(lines[n][0], lines[n][1]),
         cv::Point(lines[n][2], lines[n][3]),
         cv::Scalar(255),
         5);
// 轮廓与白线进行 AND 操作
cv::bitwise_and(contours, oneline, oneline);
```

得到的结果是一幅仅包含于指定直线相关的点。为了引入些许容忍度，我们绘制特定宽度的直线（设为 5），所有相邻范围内的点都被接收。得到的图像如图 7.13 所示，进行反转以方便观看。

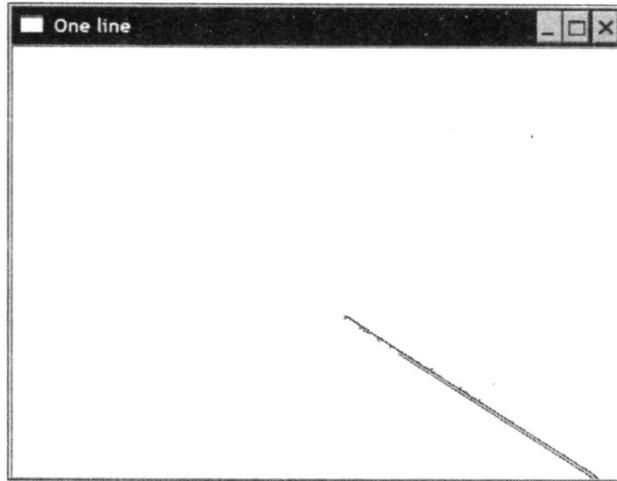


图 7.13

这个集合中的点的坐标可以在下述双重循环中置入 cv::Points 类型的 std::vector 中（也可以使用浮点类型坐标，即 cv::Point2f）：

```
std::vector<cv::Point> points;
// 遍历像素得到所有点的位置
for(int y= 0;y<oneline.rows;y++) {
    //y行
    uchar*rowPtr = oneline.ptr<uchar>(y);
    for(int x = 0;x<oneline.cols;x++) {
        //x列
        // 如果位于轮廓上
        if(rowPtr[x]){
            points.push_back(cv::Point(x,y));
        }
    }
}
```

简单地调用 cv::fitLine 函数便可以找到最合适的线：

```
cv::Vec4f line;
cv::fitLine(cv::Mat(points),line,
            CV_DIST_L2,    // 距离类型
            0,             // L2 距离不使用该参数
            0.01,0.01);   // 精确值
```

直线方程的参数以单元方向向量 (cv::Vec4f 的前两项) 及直线上一点的坐标 (cv::Vec4f 后两项) 的形式给出。对于我们的例子，反向向量是 (0.83, 0.55)，点坐标是 (366.1, 289.1)。函数的最后两个参数确定了所需要的精度。输入点的坐标原先包含在 std::vector 中，因为函数需要被转化为 cv::Mat。

通常而言，直线方程将用于计算一些属性（很好的一个例子是校正，它要求精确的参数化描述）。为了确保计算得到正确结果，我们在图像上绘制估计的直线，绘制黑色的线段，长度为 200px，宽度为 3px。

```
int x0= line[2];           // 直线上的点
int y0= line[3];
int x1= x0-200*line[0];// 使用单元向量
int y1= y0-200*line[1];// 添加长度为 200 的向量
image= cv::imread("../road.jpg",0);
cv::line(image,cv::Point(x0,y0),cv::Point(x1,y1),
         cv::Scalar(0),3);
```

结果如图 7.14 所示。

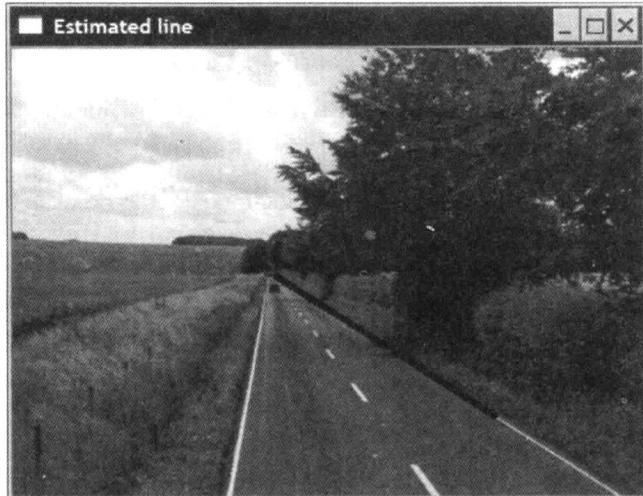


图 7.14

作用原理

直线拟合是数学中的经典问题。OpenCV 通过最小化每个点到直线的距离之和进行求解。有多个距离函数，最快速的选项是欧氏距离，即参数 CV_DIST_L2。该选项对应于标准的最小二乘法。当不属于该线的异常值可能被点集包含时，可以选择给予较远点较小的权重的距离函数。极小化过程基于 M 估计（M-estimator）技术，通过权重与点到直线距离成反比，迭代地求解加权的最小二乘问题。

使用该函数，能够拟合三维的点集。输入是一组 cv::Point3i 或 cv::Point3f 点，输出则是一个 std::Vec6f。

扩展阅读

函数 cv::fitEllipse 使用一组二维点拟合一个椭圆。它返回一个带旋转的矩形（cv::RotatedRect 实例），它内切了一个椭圆。在这个案例中，你可以这么调用：

```
cv::RotatedRect rrect= cv::fitEllipse(cv::Mat(points));
cv::ellipse(image,rrect,cv::Scalar(0));
```

你可以使用函数 cv::ellipse 来绘制得到的椭圆。

7.5 提取连通区域的轮廓

图像中通常包含着物体的表现，图像分析的一大目标便是识别并提取出这些物体。在物体检测、识别应用中，第一步是生成包含特定物体位置的二值图像。不论这个二值图像的获取方式如何（例如，可通过第4章描述的直方图反投影，或是使用我们将在第10章学习的运动分析），下一步从1和0的表示方式中提取出物体。在下面例子中，我们将使用第5章操作过的水牛图像（图7.15）。

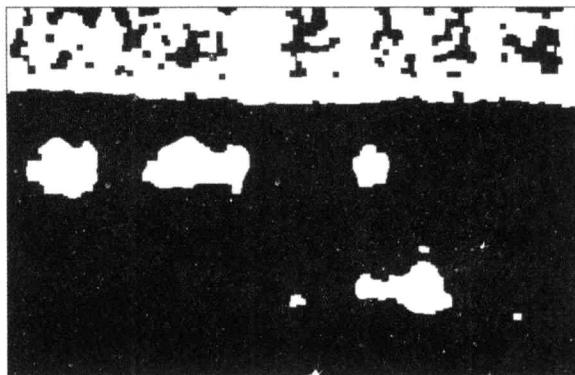


图 7.15

我们通过运用简单的阈值化操作，以及形态学开运算和闭运算获得这幅图像。本则秘诀将展示如何提取类似图像中的无图，更明确地说，我们将提取连通区域，即由二值图像中相连像素组成的形状。

实现方法

OpenCV 提供了一个简单的函数用于提取连通区域轮廓，即 `cv::findContours`：

```
std::vector<std::vector<cv::Point>>contours;
cv::findContours(image,
    contours, // 轮廓的数组
    CV_RETR_EXTERNAL, // 获取外轮廓
    CV_CHAIN_APPROX_NONE); // 获取每个轮廓的每个像素
```

输入端是一幅二值图像，而输出的是轮廓的向量，而轮廓本身又是 `cv::Points` 的向量。这解释了为何输出参数定义为 `std::vector` 的 `std::vector`。另外还涉及两个参数，第一个标记了仅返回外部轮廓，即物体的洞将会忽视（“扩展阅读”中将

介绍更多选项)。第二个参数将给出更复杂的链式轮廓近似，以得到更紧密的表示方式。对于前一幅图像，我们得到9个轮廓，这可由 `contours.size()` 的值给出。幸运的是，我们可以使用非常便利的函数来绘制这些轮廓(此处图像的背景色为白色)：

```
// 在白色图像上绘制黑色轮廓  
cv::Mat result(image.size(), CV_8U, cv::Scalar(255));  
cv::drawContours(result, contours,  
-1, // 绘制所有轮廓  
cv::Scalar(0), // 颜色为黑色  
2); // 轮廓线的绘制宽度为2
```

如果第三个参数为负值，那么将绘制所有轮廓。否则，可以指定需要绘制的轮廓的索引值。结果如图 7.16 所示。

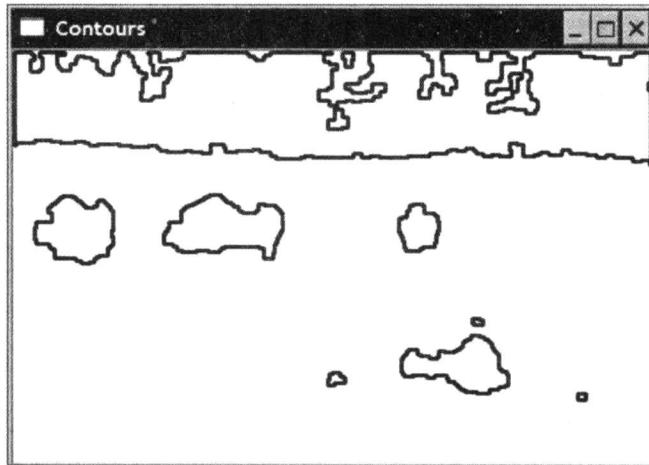


图 7.16

作用原理

轮廓的简单提取算法如下，系统性地扫描图像直到遇到连通区域的一个点，以它为起始点，跟踪它的轮廓，标记边界上的像素。当轮廓完整闭合，扫描回到上一个位置，直到再次发现新的成分。

之后识别到的连通区域可以单独分析。如果我们事先知道目标物体的期望大小，那么可以移除掉其中一些区域。我们可以使用极大极小值来限定区域的周长，可以遍历所有轮廓以去除无效的区域。

```
// 移除过长或过短的轮廓
int cmin= 100;      // 最小轮廓长度
int cmax= 1000;     // 最大轮廓长度
std::vector<std::vector<cv::Point>>::
    const_iterator itc= contours.begin();
while(itc!= contours.end()){
    if(itc->size()<cmin||itc->size()>cmax)
        itc= contours.erase(itc);
    else
        ++itc;
}
```

`std::vector` 中每次 `erase` 操作的复杂度都是 $O(N)$ ，因此这个循环可以优化得更高效。但是考虑到该向量的尺寸，这个操作不会太耗时，这次我们绘制剩余的轮廓，并得到图 7.17 所示的结果。

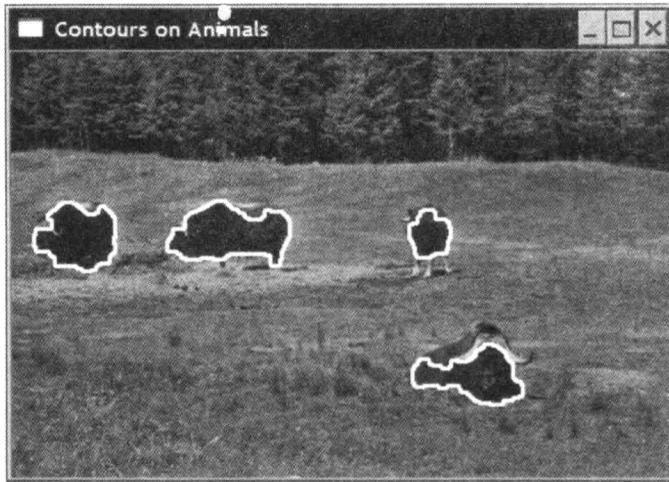


图 7.17

足够幸运的是，我们找到一个简单的标准以识别所有感兴趣的物体。在更复杂的场景中，需要更高级的属性分析方法，而这将是下一则秘诀的任务。

扩展阅读

`cv::findContours` 函数能够包含所有闭合的轮廓，甚至是由洞组成的轮廓。这可以通过指定不同的标志实现：

```
cv::findContours(image,
    contours, // 轮廓的数组
    CV_RETR_LIST, // 获取外轮廓
    CV_CHAIN_APPROX_NONE); // 获取每个轮廓的每个像素
```

通过这次调用，我们得到图 7.18 所示的轮廓。

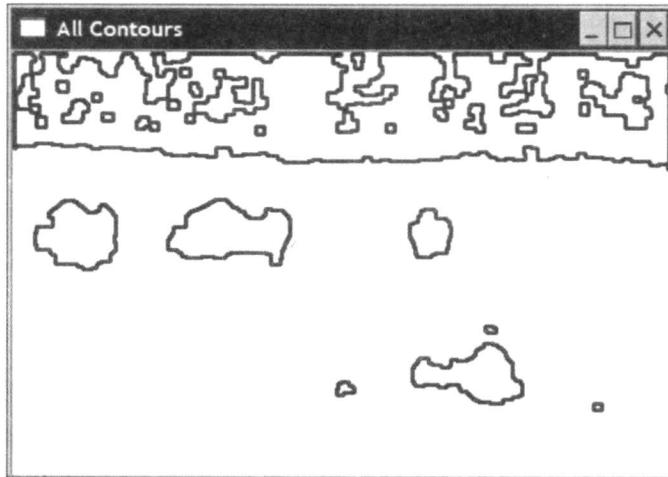


图 7.18

注意，额外的轮廓被添加到背景的森林中。也可以把这些轮廓组织到一个层次结构中，主成分是父亲，图像中它的洞是后代，如果这些洞中还存在新的连通区域，那么又将成为之前后代的后代，依此类推。这个层次结构通过使用 CV_RETR_TREE 参数获取，使用方法如下：

```
std::vector<cv::Vec4i> hierarchy;
cv::findContours(image,
    contours, // 轮廓的数组
    hierarchy, // 分层表示
    CV_RETR_TREE, // 以树状结构获取轮廓
    CV_CHAIN_APPROX_NONE); // 获取每个轮廓的每个像素
```

在这个案例中，每个轮廓都有对应的层级元素，由四个整数组成一个索引值。头两个整数给出同一年级中下一个轮廓及前一个轮廓的索引值，而后两个整数则是第一个后代及父亲的索引值。一个负的索引值意味着轮廓列表的末尾。参数 CV_RETR_CCOMP 的结果类似，但是限制层数为 2。

7.6 计算连通区域的形状描述符

连通区域通常对应于场景中的某个物体，为了识别这个物体，或者将它与其他图形元素作比较，需要进行一些测量仪获取它的特征。在这则秘诀中，我们将研究OpenCV中可用的一些形状描述符。

实现方法

许多函数可用于描述形状，我们将运用其中一些到先前提取的连通区域上。尤其是，我们向量中包含的四个轮廓对应于之前识别到的四头水牛。在下面的代码段中，我们对轮廓（`contours[0]~contours[3]`）计算形状描述符，并在轮廓图像（厚度为1）之上绘制结果（厚度为2），结果显示在本节的最后。

第一个是包围盒，应用到右下角的连通区域上：

```
// 测试包围盒  
cv::Rect r0= cv::boundingRect(cv::Mat(contours[0]));  
cv::rectangle(result,r0,cv::Scalar(0),2);
```

最小包围圆类似，应用到右上角：

```
// 测试最小包围圆  
float radius;  
cv::Point2f center;  
cv::minEnclosingCircle(cv::Mat(contours[1]),center,radius);  
cv::circle(result,cv::Point(center),  
           static_cast<int>(radius),cv::Scalar(0),2);
```

轮廓的多边形近似被应用到左边的连通区域上：

```
// 测试多边形近似  
std::vector<cv::Point>poly;  
cv::approxPolyDP(cv::Mat(contours[2]),poly,  
                  5,           // 近似的精确度  
                  true);     // 这是个闭合形状
```

在图像上绘制结果需要更多工作：

```
// 遍历每个片段进行绘制
std::vector<cv::Point>::const_iterator itp= poly.begin();
while(itp!=(poly.end()-1)){
    cv::line(result,*itp,*(itp+1),cv::Scalar(0),2);
    ++itp;
}
// 首尾用直线相连
cv::line(result,
        *(poly.begin()),
        *(poly.end()-1),cv::Scalar(20),2);
```

凸包 (Convex Hull) 是另一种多边形近似：

```
// 测试凸包
std::vector<cv::Point> hull;
cv::convexHull(cv::Mat(contours[3]),hull);
```

最终，力矩 (Moment) 的计算是另一种强大的描述符：

```
// 测试力矩
// 遍历所有轮廓
itc= contours.begin();
while(itc!=contours.end()){
    // 计算所有的力矩
    cv::Moments mom= cv::moments(cv::Mat(*itc++));
    // 绘制质心
    cv::circle(result,
               // 质心坐标转换为整数
               cv::Point(mom.m10/mom.m00,mom.m01/mom.m00),
               2,cv::Scalar(0),2); // 绘制黑点
}
```

生成的图像如图 7.19 所示。

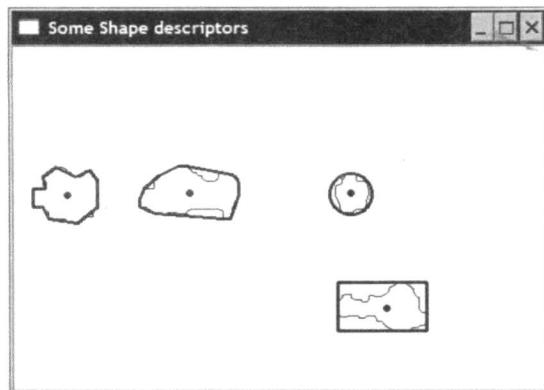


图 7.19

作用原理

包围盒也许是最紧密的展现及定位连通区域的方式，它被定义为正好包含形状的最小矩形。比较盒子的高度和宽度能够给出物体的垂直或水平方向（例如，可以区分汽车与行人）。最小包围圆通常用于仅需要尺寸与位置的情况。

多边形近似在需要操作与形状类似的更紧密的表现时很有用，需要指定精确值参数以确定形状与它的简化多边形之间的最大可接受距离，这是 `cv::approxPolyDP` 函数的第四个参数。结果是 `cv::Point` 的向量，对应多边形的顶点。为了绘制该多边形，我们需要遍历向量，将相邻的点连接以绘制它们之间的直线。

凸包，或凸包络，是包围形状的最小凸多边，它可以被可视化为围绕连通区域的橡皮筋形成的形状。

力矩通常用于以数学方式进行形状的结构化分析。OpenCV 定义了一种数据结构，封装了所有计算而得的力矩，即 `cv::moments` 函数的返回值。我们用该函数计算每个连通区域的前三个空间力矩，以获得质心。

扩展阅读

OpenCV 还提供了其他的结构化属性计算方式。函数 `cv::minAreaRect` 计算最小包围旋转矩形，函数 `cv::contourArea` 计算轮廓的面积（包含的像素个数），函数 `cv::pointPolygonTest` 确定一个点是否位于轮廓内，而 `cv::matchShapes` 测量两个轮廓间的相似度。

第 8 章

检测并匹配兴趣点

本章主要探讨：

- ◆ 检测 Harris 角点；
- ◆ 检测 FAST 特征；
- ◆ 检测尺度不变的 SURF 特征；
- ◆ 描述 SURF 特征。

8.1 引言

在计算机视觉中，兴趣点（也叫做关键点或特征点）的概念被大量用于解决物体识别、图像匹配、视觉跟踪、三维重建等问题。它依赖于这个想法，即不再观察整幅图像，而是选择某些特殊的点，然后对它们执行局部分析。如果能检测到足够多的这种点，同时它们的区分度很高，并且可以精确定位稳定的特征，那么这个方法就很有效。本章将介绍一些特征点检测器以及它们在图像匹配中的应用。

8.2 检测Harris角点

在图像中检测兴趣点时，角点可以作为一个有趣的方案。它们在图像中可以轻易地定位，同时，它们在人造物体的场景（比如墙、门、窗、桌等）中随处可见。角点的意义还在于它们是可以精确定位（甚至可以达到亚像素的精度）的二维特征，因为它们位于两条边缘的交点处。这与位于相同强度区域的点不同，与物体轮廓上的点也不同，轮廓点难以在其他图形的相同物体上精确定位。

Harris 特征检测器是一种经典的角度检测方法，我们将在本则秘诀中探讨它。

实现方法

OpenCV 中用于实现 Harris 角点检测的基本函数时，`cv::cornerHarris` 使用非常简便。输出结果为浮点数类型的图像，其中每一项为对应位置的角点强度。之后可使用阈值化以得到一组检测到的角点，调用方式如下：

```
// 检测 Harris 角点
cv::Mat cornerStrength;
cv::cornerHarris(image,cornerStrength,
                 3,           // 相邻像素的尺寸
                 3,           // 滤波器的孔径大小
                 0.01);      // Harris 参数

// 角点强度的阈值
cv::Mat harrisCorners;
double threshold= 0.0001;
cv::threshold(cornerStrength,harrisCorners,
               threshold,255,cv::THRESH_BINARY);
```

原图如图 8.1 所示。

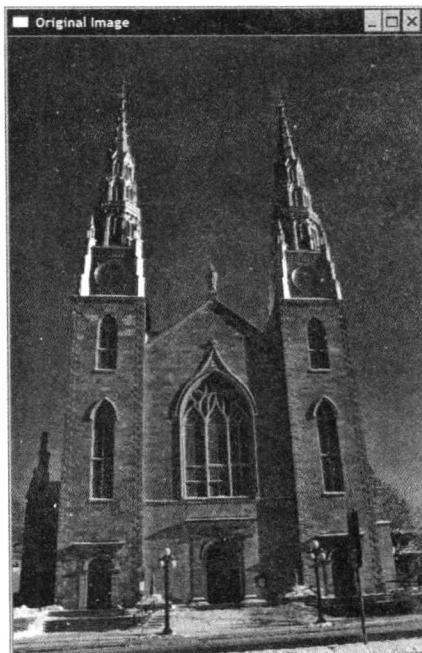


图 8.1

得到的结果为二值图像，加以反转以方便观看（使用 `cv::THRESH_BINARY_INV` 而不是 `cv::THRESH_BINARY`，使得角点为黑色），如图 8.2 所示。

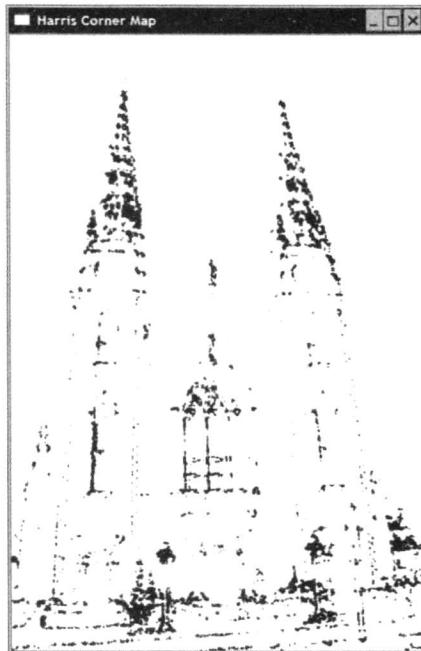


图 8.2

从之前的函数调用中，我们观察到这个特征点检测器需要多个参数，因此难以调优。此外，获取的角点图形包含许多角点群，这与我们需要检测精确定位点的事实矛盾。因此，我们将通过封装自定义类来改进角点检测的方法。

该类封装了 Harris 参数，提供默认值，以及相应的设置函数与取值函数：

```
class HarrisDetector{  
private:  
    // 表示角点强度的 32 位浮点图像  
    cv::Mat cornerStrength;  
    // 表示阈值化后角度的 32 位浮点图像  
    cv::Mat cornerTh;  
    // 局部极大值图像（内部）  
    cv::Mat localMax;  
    // 导数平滑的相邻像素的尺寸  
    int neighbourhood;  
    // 梯度计算的孔径大小
```

```
int aperture;
//Harris 参数
double k;
// 阈值计算的最大强度
double maxStrength;
// 计算得到的阈值（内部）
double threshold;
// 非极大值抑制的相邻像素的尺寸
int nonMaxSize;
// 非极大值抑制的核
cv::Mat kernel;

public:
    HarrisDetector():neighbourhood(3),aperture(3),
                      k(0.01),maxStrength(0.0),
                      threshold(0.01),nonMaxSize(3){
        // 创建非极大值抑制的核
        setLocalMaxWindowSize(nonMaxSize);
    }
```

为了检测 Harris 角点，我们使用两个步骤。首先，计算每个像素处的 Harris 值：

```
// 计算 Harris 角点
void detect(const cv::Mat& image){
    //Harris 计算
    cv::cornerHarris(image,cornerStrength,
                      neighbourhood, //neighborhood size
                      aperture,      // 滤波器的孔径大小
                      k);           //Harris 参数

    // 内部阈值计算
    double minStrength;//未使用
    cv::minMaxLoc(cornerStrength,
                  &minStrength,&maxStrength);

    // 局部极大值检测
    cv::Mat dilated; //临时图像
    cv::dilate(cornerStrength,dilated,cv::Mat());
    cv::compare(cornerStrength,dilated,
                localMax,cv::CMP_EQ);

}
```

接着，基于特定的阈值获取特征点。由于 Harris 可能的取值范围依赖于特定参数的组合，阈值被设置为计算得到的最大 Harris 值的部分：

```
// 由 Harris 值获取角点图
cv::Mat getCornerMap(double qualityLevel) {
    cv::Mat cornerMap;
    // 对角点图像进行阈值化
    threshold= qualityLevel*maxStrength;
    cv::threshold(cornerStrength,cornerTh,
                  threshold,255,cv::THRESH_BINARY);
    // 转换为 8 位图像
    cornerTh.convertTo(cornerMap,CV_8U);
    // 非极大值抑制
    cv::bitwise_and(cornerMap,localMax,cornerMap);
    return cornerMap;
}
```

该方法返回二值的角点图像，Harris 特征的检测被划分为两个函数，这使得我们可以避免耗时的运算，尝试不同的阈值进行检测（直到获取合适数量的特征点），同时能够以 `cv::Point` 类型 `std::vector` 的形式获取 Harris 特征：

```
// 由 Harris 值得到特征点
void get Corners (std::vector<cv::Point>& points,
                  double qualityLevel) {
    // 得到角点图
    cv::Mat cornerMap= getCornerMap(qualityLevel);
    // 得到角点
    get Corners (points,cornerMap);
}

// 由角点图获取特征点
void get Corners (std::vector<cv::Point>& points,
                  const cv::Mat& cornerMap) {
    // 遍历像素得到所有特征
    for(int y=0;y<cornerMap.rows;y++){
        const uchar*rowPtr = cornerMap.ptr<uchar>(y);
        for(int x= 0;x<cornerMap.cols;x++){
            // 如果是特征点
            if(rowPtr[x]){

```

```
    points.push_back(cv::Point(x,y));
}
}
}
}
```

该类通过增加非极大值抑制步骤改进了 Harris 角点检测的效果。检测到的点通过 cv::circle 函数得到绘制：

```
// 在特征点的位置绘制圆
void drawOnImage(cv::Mat &image,
                 const std::vector<cv::Point>&points,
                 cv::Scalar color= cv::Scalar(255,255,255),
                 int radius=3,int thickness=2){
    std::vector<cv::Point>::const_iterator it=
        points.begin();
    // 对于所有角点
    while(it!=points.end()){
        // 绘制一个圆
        cv::circle(image,*it,radius,color,thickness);
        ++it;
    }
}
```

使用该类时的检测过程如下：

```
// 创建 Harris 检测器的对象
HarrisDetector harris;
// 计算 Harris 值
harris.detect(image);
// 检测 Harris 角点
std::vector<cv::Point>pts;
harris.getorners(pts,0.01);
// 绘制 Harris 角点
harris.drawOnImage(image,pts);
```

生成的图像如图 8.3 所示。



图 8.3

作用原理

为了定义图像中的角点，Harris 观察一个假定的特征点周围小窗口内的方向性强度平均变化。如果我们考虑偏移向量 (u, v) ，它的平均强度变化为

$$R = \sum (I(x+u, y+v) - I(x, y))^2$$

求和过程覆盖了预定义的相邻像素点（相邻像素的尺寸对应于 `cv::cornerHarris` 函数中的第三个参数）。强度变化的平均值在所有可能的方向进行计算，因为高强度变化会出现在多个方向。通过这个定义，Harris 测试的过程如下。我们首先获取平均强度变化最大值对应的方向，接着检查位于它垂直方向的变化是否也很强烈，同时满足条件的便是一个角点。

用数学术语讲，测试这个条件时可使用泰勒级数展开对先前公式进行近似：

$$R \approx \sum \left(I(x,y) + \frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v - I(x,y) \right) = \sum \left[\left(\frac{\partial I}{\partial x} u \right)^2 + \left(\frac{\partial I}{\partial y} v \right)^2 + 2 \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} uv \right]$$

可用矩阵形式重写：

$$R \approx [u \ v] \begin{bmatrix} \Sigma \left(\frac{\delta I}{\delta x} \right)^2 & \Sigma \frac{\delta I}{\delta x} \frac{\delta I}{\delta y} \\ \Sigma \frac{\delta I}{\delta x} \frac{\delta I}{\delta y} & \Sigma \left(\frac{\delta I}{\delta y} \right)^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

这是一个协方差（Covariance）矩阵，表现的是所有方向上强度的变化率。该定义涉及图像的一阶导数，这通常是 Sobel 算子的计算结果，在 OpenCV 的函数实现中第四个参数对应是用于计算 Sobel 滤波器的孔径（Aperture）。协方差矩阵的两个特征值（Eigenvalue）给出了最大平均强度变化以及垂直方向上的平均强度变化，如果这两个特征值比较低，那么我们位于相对同质（Homogenous）的区域。如果其中一个较高，另一个较低，那么我们一定位于边上。最后，如果两个特征值都较高，那么我们位于角点处。因此，角点成立的条件是拥有超过阈值的协方差矩阵的最小特征值。

Harris 角点算法的原始定义使用特征分解理论中的一些特性以避免计算特征值，这些特性包括：

- ◆ 矩阵的特征值之积等于它的行列式（Determinant）；
- ◆ 矩阵的特征值之和等于矩阵对角线之和，即矩阵的迹（Trace）。

然后，我们可以通过计算下述分数验证两个特征值是否为足够高：

$$\text{Det}(C) - k \cdot \text{Trace}^2(C)$$

我们可以轻易验证只在两个特征值都为高时才能得高分，这正是 `cv::cornerHarris` 函数在每个像素位置计算得到的分数。 k 的值指定为函数的第五个参数，也许很难确定最优值。然而，在实践中，0.05 ~ 0.5 之间的值通常能产生满意的结果。

为了改进检测结果，该类添加了额外的非极大值抑制步骤，目的是移除彼此相邻的 Harris 角点。因此，为了最终被接受，Harris 角点只需要得分高于给定阈值，它还必须是局部极大值。测试时使用简单的技巧，包括在检测函数中将 Harris 得分的图像进行膨胀：

```
cv::dilate(cornerStrength, dilated, cv::Mat());
```

由于膨胀运算替换每个像素值为相邻范围内的最大值，只有局部极大值的点才会保留原样，这可以通过下述相等测试进行验证：

```
cv::compare(cornerStrength,dilated,localMax,cv::CMP_EQ);
```

localMax 矩阵仅在局部极大值的位置为真（即非零），我们在 getCornerMap 函数中用它来抑制所有非极大值的特征（基于 cv::bitwise_and 函数）。

扩展阅读

额外的改进也可应用于原始的 Harris 角点算法，本节将描述 OpenCV 实现的其他角点检测器，它们扩展 Harris 检测器使得角点的分布更加均匀。我们将看到，这个运算子在 OpenCV 2 用于特征检测的通用接口中得到实现。

1. 适合跟踪的优质特征

在浮点处理器的帮助下，为了避免特征值分解而引入的数学上的简化变得微不足道，因此 Harris 检测可以基于计算而得的特征值。原则上，这个修改不会显著影响检测的结果，但是能够避免使用任意的 k 参数。

第二个修改用于解决特征点聚类的问题，除了引入局部极大值的条件，特征点倾向于在图像中不均匀分布，集中在纹理丰富的部分。一种解决方案是利用两个特征点之间的最小距离，这正是下述算法将要实现的。我们从 Harris 得分最高的点开始（意味着它有着最大的最小特征值），仅接受离开有效特征点距离大于特定值的点。这便是 cv::goodFeaturesToTrack 实现的方法，它检测到的特征能用于视觉跟踪应用中的优质特征集合，因此函数叫这个名字。它的调用方式如下：

```
// 计算适合跟踪的优质特征
std::vector<cv::Point2f>corners;
cv::goodFeaturesToTrack(image,corners,
500,      // 返回的最大特征点数目
0.01,     // 质量等级
10);      // 两点之间的最小允许距离
```

除了质量等级阈值、特征点之间的最小允许距离，该函数还需要指定返回的最大特征点数目（因为特征点是按照强度进行排序的），先前的函数生成的结果如图 8.4 所示。

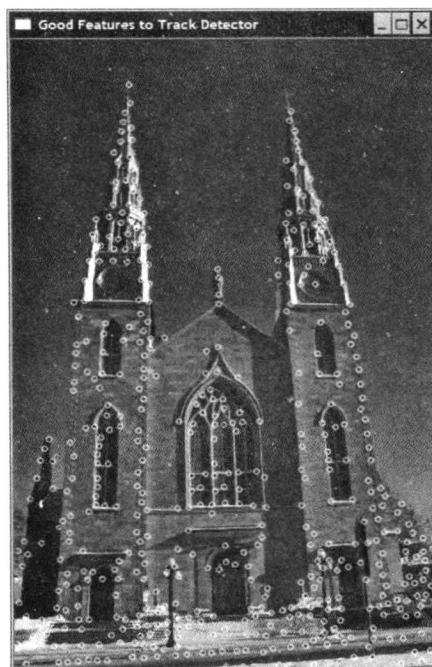


图 8.4

该方法增加了检测的复杂度，因为要求特征点按照它们的 Harris 得分进行排序，但是它也显著改进了特征点的分布情况。该函数也可指定一个可选的参数，使得按照经典的角度分数定义进行计算（即使用协方差矩阵的行列式与迹）。

2. 特征检测器通用接口

OpenCV 2 中的通用特征检测器引入了一种新的通用接口用于不同的检测器，该接口允许在同一个应用中简单测试不同的检测器。

该接口定义了一个 `KeyPoint` 类以封装每个特征点的属性，对于 Harris 角点，只有位置是有用的。秘诀“检测尺度不变的 SURF 特征”中将讨论其他可用的属性。抽象类 `cv::FeatureDetector` 中提供了不同签名的检测函数：

```
void detect(const Mat& image, vector<KeyPoint>& keypoints,
            const Mat& mask=Mat()) const;
void detect(const vector<Mat>& images,
            vector<vector<KeyPoint>>& keypoints,
            const vector<Mat>& masks=vector<Mat>() const;
```

第二个函数运行特征点在图像向量中进行检测，该类也包含在文件中读写特征点的函数。

`cv::goodFeaturesToTrack` 函数拥有一个封装类 `cv::GoodFeatureToTrackDetector`，它继承自 `cv::FeatureDetector` 类。它的用法与我们自定义的 Harris 检测类相似，即：

```
// 特征点向量
std::vector<cv::KeyPoint> keypoints;
// 检测器的构造函数
cv::GoodFeaturesToTrackDetector gftt(
    500,      // 返回的最大特征点数目
    0.01,     // 质量等级
    10);      // 两点之间的最小允许距离
// 使用 FeatureDetector 的函数进行检测
gftt.detect(image, keypoints);
```

结果与先前得到的一致，因为它们调用的是同一个函数。

参 考

描述 Harris 算子的经典论文 “*A Combined Corner and Edge Detector*” (C.Harris, M.J. Stephens, *Alvey Vision Conference*, pp.147–152, 1988)。

J.Shi 及 C.Tomasi 撰写的论文 “*Good Features to Track*” (*Int.Conference on Computer Vision and Pattern Recognition*, pp.593-600, 1994) 论述了这些特征。

K.Mikolajczyk 及 C.Schmid 撰写的论文 “*Scale and Affine Invariant Interest Point Detectors*” (*International Journal of Computer Vision*, Vol.60, No.1, pp.63-86, 2004)，提出了多尺度及放射不变的 Harris 算子。

8.3 检测FAST特征

Harris 算子提出了角点（或者说特征点）的正式数学定义，它基于两个正交方向上的强度变化率。尽管这是一个合理的定义，它需要耗时地计算图像的导数，尤其是考虑到特征点检测只是复杂算法的第一步。

本则秘诀中，我们展示另一种检测算子，它经过特别设计，能快速进行检测，依赖少数像素比较来确定是否接受一个特征点。

实现方法

我们使用 OpenCV 2 的通用接口来创建任意的特征检测器，这里使用的是 FAST 检测器。顾名思义，它的计算非常快速：

```
// 特征点的向量  
std::vector<cv::KeyPoint> keypoints;  
// 构造 FAST 特征检测器  
cv::FastFeatureDetector fast(  
    40); // 检测阈值  
// 进行检测  
fast.detect(image, keypoints);
```

OpenCV 同时提供一个通用的特征点绘制函数：

```
cv::drawKeypoints(image, // 原始图像  
    keypoints, // 特征点向量  
    image, // 输出图像  
    cv::Scalar(255,255,255), // 特征点颜色  
    cv::DrawMatchesFlags::DRAW_OVER_OUTIMG); // 绘制标记
```

通过指定选中的绘制标记，关键点在输出图像上进行绘制，生成图 8.5 所示的结果。

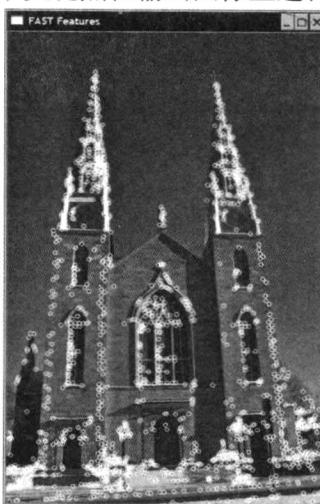


图 8.5

设置特征点颜色时可以赋予一个负值，这将产生有趣的结果，即绘制的圆将拥有不同的随机颜色。

作用原理

与 Harris 算法的情况相同的是，FAST (Features from Accelerated Segment Test) 特征算法需要定义什么是“角点”。这次的定义基于假定特征点周围的图像强度，通过检查候选像素周围一圈像素来决定是否接受一个特征点。与中心点差异较大的像素如果组成连续的圆弧，并且弧长大于圆周长的 $3/4$ ，那么我们认为找到了一个特征点。

这是一个简单测试，同时算法使用额外技巧进行加速。如果我们首先测试圆上被 90° 分隔的四个点（比如说顶部、底部、左侧及右侧点），那么为了满足上述条件，四个点中至少三个点必须大于或小于中心像素。如果这不成立，那么该点可以被直接移除而不需要检查额外圆周上的点。在实践中，大部分的像素点可以通过该测试进行移除，因此它非常高效。

原则上，测试中的圆的半径应该是一个参数，然而实践中我们发现，半径为 3 可以兼顾结果及效率。如图 8.6 所示，需要考虑圆周上的 16px。

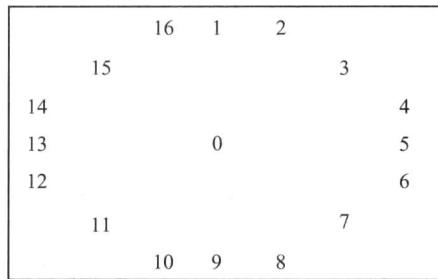


图 8.6

用于预处理的四个点是 1、5、9 和 13。

和 Harris 特征相同的是，可以在找到的角点上执行非极大值抑制，因此需要指定角点强度的测量方法。可以采用不同的方案，我们使用的是连续圆弧上像素点与中心点差值的绝对值之和。

该算法可以获得非常快速的特征点检测，在需要考虑运行速度时可以选用，如在高帧率的视频序列中进行视觉跟踪时。

参 考

E.Rosten 及 T.Drummond 撰写的论文 “Machine Learning for High-speed Corner Detection” (In European Conference on Computer Vision, pp.430-443, 2006) 细致描述了 FAST 特征算法。

8.4 检测尺度不变的SURF特征

当尝试在不同图像之间匹配特征时，我们通常面临尺度变化的难题，即需要分析的图像在拍摄时与目标物体的距离是不同的，因此，目标物体在图像中有些不同的尺寸。如果我们尝试使用固定尺寸的相邻尺寸来匹配不同图像中的相同特征，那么由于尺度的变化，它们的强度模版并不会匹配。

为了解决这个问题，计算机视觉中引入尺度不变的特征，主要的思想是每个检测到的特征点都伴随着对应的尺寸因子。近年来，人们提出了多个尺度不变的特征，本则秘诀展示其中之一，即 SURF (Speeded Up Robust Features) 特征——加速鲁棒特征。我们将看到它们不仅有尺度不变的特性，而且计算非常高效。

实现方法

SURF 特征在 OpenCV 中的实现也使用了 `cv::FeatureDetector` 接口，因此，检测的过程与先前展示过的非常类似：

```
// 特征点的向量
std::vector<cv::KeyPoint> keypoints,
// 构造 SURF 特征检测器
cv::SurfFeatureDetector surf(
    2500.); // 阈值
// 检测 SURF 特征
surf.detect(image, keypoints);
```

为了绘制这些特征，我们再次使用 `cv::drawKeypoints` 函数，但使用另一个掩码以展示每个特征的缩放因子：

```
// 绘制特征点，加上尺度与方向信息
cv::drawKeypoints(image, // 原始图像
    keypoints, // 特征点的向量
    featureImage, // 生成图像
    cv::Scalar(255, 255, 255), // 特征点的颜色
    cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS); // 标志位
```

带有详细信息的生成图像如图 8.7 所示。

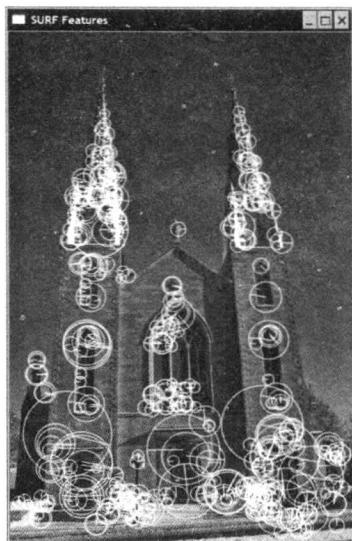


图 8.7

通过截图我们可以看到，使用 DRAW_RICH_KEYPOINTS 之后关键点上圆圈的尺寸与特征的尺度成正比。同时，SURF 算法还将方向与每个特征联系使得它们具有旋转无关性，方向在圆圈中以放射性线段的形式绘制。

如果我们对相同的物体拍摄一幅不同尺度的图像，那么检测到的特征值结果如图 8.8 所示。

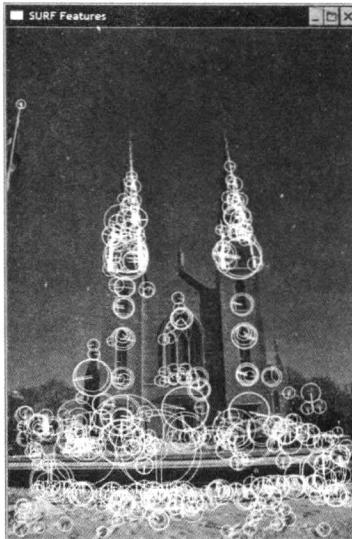


图 8.8

仔细观察检测到的特征点，可以看到对应的圆圈的尺寸与图像尺度的改变成正比。比如右上窗口中的底部，在两幅图像中都检测到了 SURF 特征，两个对应的（不同尺寸的）圆圈包含相同的视觉元素。当然，并非所有特征皆如此，我们将在下一章讨论，足够高的重复度可以带来两幅图像间良好的匹配。

作用原理

在第 6 章中，我们学到一幅图像的导数可以通过高斯滤波进行估算，这些滤波器使用参数 σ 来定义滤波核的大小。我们将看到， σ 对应于创建滤波器的高斯函数的方差（Variance），它隐式地定义了估算导数时的尺度。事实上，拥有较大 σ 值的滤波器对图像细节进行平滑，这就是为何我们说它工作于粗糙的尺度。

现在，假设我们通过不同尺度的高斯滤波计算给定图像位置的拉普拉斯算子，那么可以获取不同的值。观察滤波器在不同尺度因子下的变化，我们获得最大值位于 σ 的曲线。如果我们对不同尺度的图像提取这个最大值，这两个值的比例将对应于图像拍摄时尺度的比例。我们观察到的这个重要特性是提取尺度不变特征时的核心原理，即尺度不变特征应该既是空间域（图像中）上又是尺度域（在求导滤波器应用于不同尺度时获得）上的局部极大值。

SURF 的实现如下，首先对每个像素计算 Hessian 矩阵以得到特征，该矩阵测量一个函数的局部曲率，定义如下：

$$H(x,y) = \begin{bmatrix} \frac{\delta^2 I}{\delta x^2} & \frac{\delta^2 I}{\delta x \delta y} \\ \frac{\delta^2 I}{\delta x \delta y} & \frac{\delta^2 I}{\delta y^2} \end{bmatrix}$$

该矩阵的行列式给出曲率的强度，定义角点为具有较高局部曲率的图像点（即在多个方向具有高曲率）。由于该矩阵是由二阶导数组成的，它可以使用不同 σ 尺度的 Laplacian Gaussian 核进行计算，因此 Hessian 变成了三个变量的函数： $H(x, y, \sigma)$ 。当 Hessian 值同时在空间域和尺度域上达到局部极大值时（即需要进行 $3 \times 3 \times 3$ 的非极大值抑制），我们认为找到了尺度不变的特征。但是，这个行列式必须含有 cv::SurfFeatureDetector 构造函数中第一个参数所指定的最小值。

所有这些不同尺度的运算都很耗时，SURF 算法的目的是尽可能高效。因此我们使

用近似的高斯核，仅涉及少量整数加法，结构如图 8.9 所示。

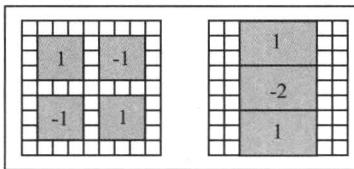


图 8.9

左侧的核用于估算混合二阶导数，右侧的核则用于估算垂直方向的二阶导数，后者的旋转版本用于估算水平方向的二阶导数。最小的核拥有 9×9 的尺寸，对应于 $\sigma \approx 1.2$ ，后续的核的尺寸不断递增。滤波器的精确数目可以通过 SURF 类的额外参数进行设置，默认情况我们使用 12 个不同尺寸的核（最大值为 99×99 ）。积分图像的使用保证了每个区域之和能够通过三次加法得到计算，同时这些加法与滤波器的尺寸无关。

一旦识别了局部极大值，每个特征点的精确位置可以通过空间域及尺度域上进行插值获取。结果是一组具有亚像素精确度的特征点，以及一个对应的尺度值。

扩展阅读

SURF 算法是著名的尺度不变特征检测器 SIFT (Scale-Invariant Feature Transform) 的高效变种，SIFT 也检测空间域及尺度域上的局部极大值作为特征，但是使用 Laplacian 滤波器响应而不是 Hessian 行列式，不同尺度的 Laplacian 通过 Gaussian 滤波器的差值进行计算。OpenCV 有一个封装类用于检测这些特征：

```
// 特征点的向量
std::vector<cv::KeyPoint> keypoints;
// 构造 SURF 特征检测器
cv::SiftFeatureDetector sift(
    0.03,      // 特征的阈值
    10.);      // 用于降低直线敏感度的阈值
// 检测 SURF 特征值
sift.detect(image, keypoints);
```

结果也非常相似，如图 8.10 所示。

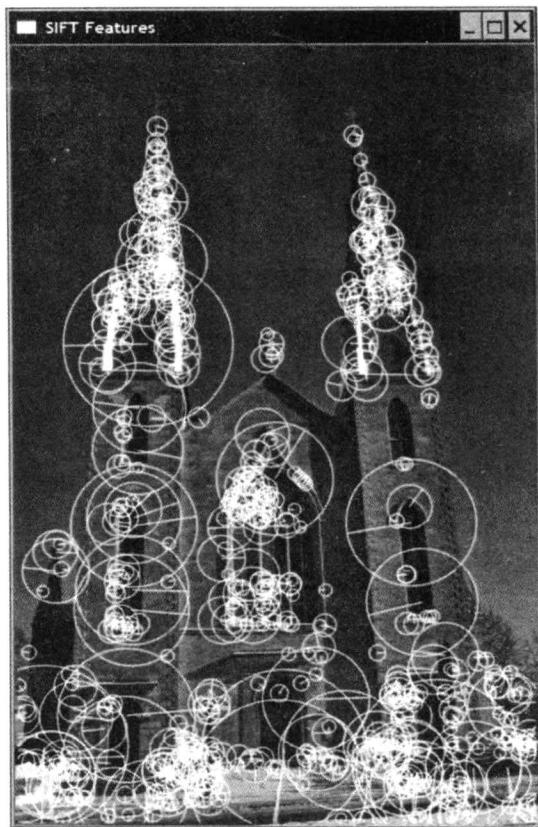


图 8.10

然而，由于特征点的计算基于浮点核，因此通常会认为它们在空间和尺度上更加精确，当然，也会更加耗时。

参 考

论 文 “SURF: Speeded Up Robust Features” (H. Bay, A. Ess, T. Tuytelaars, L.Van Gool, *Computer Vision and Image Understanding*, Vol.110, No.3, pp.346-359, 2008) 描述了 SURF 特征。

D.Lowe 撰写的开天辟地之作 “Distinctive Image Features from Scale Invariant Features” (*International Journal of Computer Vision*, Vol.60, No.2, pp.91-110, 2004) 描述了 SIFT 算法。

8.5 描述SURF特征

SURF 算法为每个检测到的特征定义了位置和尺度，尺度值可用于定义围绕特征点的窗口大小，不论物体的尺度如何窗口中都将包含相同的视觉信息，这些信息将用于表示特征点以使得它们与众不同。

本则秘诀将展示如何使用简洁描述特征点的相邻窗口，在特征匹配中，特征描述子通常是 N 维向量，在光照不变以及少许透视形变的情况下很理想。另外，优质的描述子可以通过简单的距离测量进行比较（如欧氏距离）。因此，它们在特征匹配算法中的作用非常大。

实现方法

下述代码与之前用于特征检测的非常相似，OpenCV 2 中引入了一个通用类，用于提取不同的特征点描述子。为了延续之前的秘诀，我们使用适合 SURF 算法的描述子。基于特征检测环节获取的 `cv::Keypoint` 向量，描述子的计算方式如下：

```
// 构造 SURF 描述子提取器
cv::SurfDescriptorExtractor surfDesc;
// 提取 SURF 描述子
cv::Mat descriptors1;
surfDesc.compute(imagel, keypoints1, descriptors1);
```

结果是一个矩阵（即 `cv::Mat` 实例），它的行数与特征点向量中的元素个数一致。每行都是一个 N 维描述子的向量，比如 SURF 算法默认的描述子维度为 64，该向量描绘了特征点周围的强度样式。两个特征点越相似，它们的特征向量也越靠近。

这些描述子在图像匹配中尤其有用，如我们想匹配同一个场景中的两幅图像。首先，我们检测每幅图像中的特征，然后提取它们的描述子。第一幅图像中的每个特征描述子向量都会与第二幅图像中的描述子进行比较，得分最高的一对描述子（即两个向量的距离最近）将视为那个特征的最佳匹配。该过程对于第一幅图像中的所有特征进行重复，这便是 `cv::BruteForceMatcher` 中实现的最基本策略。它的使用如下：

```
// 构造匹配器
cv::BruteForceMatcher<cv::L2<float>>matcher;
// 匹配两幅图像的描述子
std::vector<cv::DMatch>matches;
matcher.match(descriptors1, descriptors2, matches);
```

该类是 `cv::DescriptorMatcher` 的一个子类，前者定义了不同的匹配策略的共同接口，结果是一个 `cv::DMatch` 向量，它将用于表示一对匹配的描述子。`cv::DMatch` 数据结构包含指向第一个向量的索引，以及指向第二个向量的索引，同时包含一个表示描述子距离的浮点数，它被用于 `cv::DMatch` 实例的操作符 < 比较方法。

为了让匹配操作的结果可视化，OpenCV 提供一个绘制函数以生产由两幅输入图像衔接而成的图像，匹配的点由直线相连。在先前的秘诀中，第一幅图像检测到 340 个 SURF 特征点，蛮力法将生成相同数目的匹配。绘制所有直线将使得结果不具可读性，因此，我们将仅显示距离最近的 25 个匹配结果。这通过 `std::nth_element` 得到实现，它将第 n 个元素按照排序方式放置在第 n 个位置，所有小于它的元素都位于它之前，然后向量中剩余的元素被移除：

```
std::nth_element(matches.begin(),           // 初始位置
                  matches.begin() + 24,      // 排序元素的位置
                  matches.end());          // 终止位置
// 移除第 25 位之后所有的元素
matches.erase(matches.begin() + 25, matches.end());
```

先前的代码可以工作，因为操作符 < 已经在 `cv::DMatch` 中得到定义，这 25 个匹配因此可以得到可视化：

```
cv::Mat imageMatches;
cv::drawMatches(
    image1, keypoints1, // 第一幅图像及其特征点
    image2, keypoints2, // 第二幅图像及其特征点
    matches,           // 匹配结果
    imageMatches,       // 生成的图像
    cv::Scalar(255, 255, 255)); // 直线的颜色
```

生成的图像如图 8.11 所示。

我们可以看到，大多数的匹配正确地将左侧特征点与对应的右侧点相连，也会观察到一些错误，这是因为建筑物对称的外观使得一些局部匹配产生歧义（顶端的匹配便是一个匹配错误的例子）。

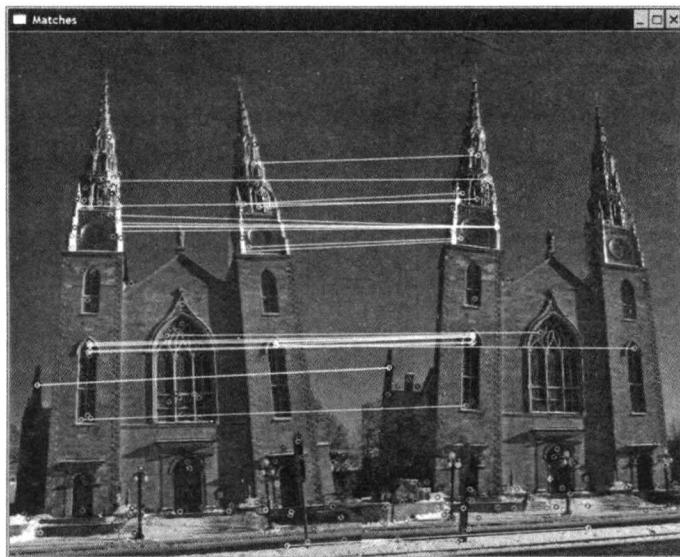


图 8.11

作用原理

优质的特征描述子必须在光照和视角发生轻微变化，并且受到图形的噪点影响下，具有不变性。因此，它们通常基于局部强度的差值，SURF 描述子将在特征点周围较大范围内应用图 8.12 所示的简单核。

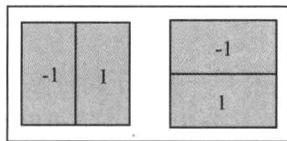


图 8.12

第一个核仅仅测量水平方向的局部强度差值（记为 dx ），而第二个核测量垂直方向的差值（记为 dy ）。用于提取描述子向量的相邻区域的尺寸定义为特征尺度因子的 20 倍（即 20σ ）。方形区域被分为 4×4 大小的较小子区域，对于每个子区域我们计算 5×5 整齐摆放的 dx 响应和 dy 响应（核的尺寸为 2σ ）。对所有这些响应求和，对每个子区域提取 4 个描述子的值：

$$[\Sigma dx \quad \Sigma dy \quad \Sigma |dx| \quad \Sigma |dy|]$$

由于存在 $4 \times 4=16$ 个子区域，我们总共有 64 个描述子的值，为了给予靠近的像素值更多的的重要性，核响应按照中心位于特征点位置的高斯函数进行加权 ($\sigma=3.3$)。

dx 及 dy 响应同时用于估算特征的方向，这些值在半径为 6σ 的圆形范围内进行计算(核的尺寸为 4σ)，位置按照 σ 的间隔进行排列。对于给定的方向，特定角度间隔($\pi/3$)内的响应进行求和，给出最长向量的方向即为主要方向。

使用 SURF 特征与描述子，可以实现尺度不变的匹配，图 8.13 展示了两幅不同尺度图像之间的 25 对最佳匹配。

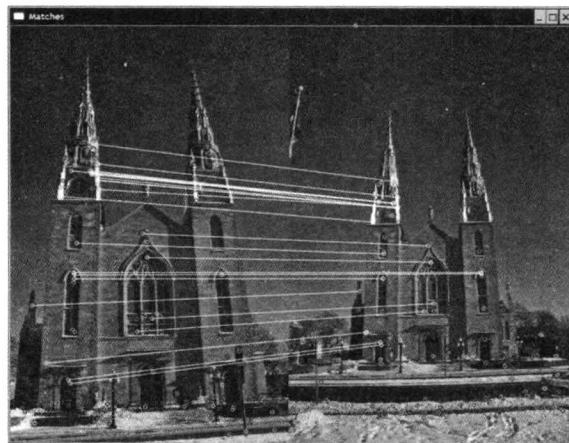


图 8.13

扩展阅读

SIFT 算法也定义了自己的描述子，它基于特征点的尺度下计算的梯度大小及方向。SURF 描述子缩放后的相邻区域被划分为 4×4 个子区域，对于其中每一个区域，我们构建 8 个元素的梯度方向直方图（受到他们的大小以及一个集中于特征点位置的全局高斯窗口的影响）。因此，描述子向量由直方图的元素组成，由于是 4×4 区域及 8 个元素的直方图，最终描述子的长度为 128。

在特征检测中，SURF 及 SIFT 描述子的区别主要是速度及精度。SUFT 描述子大部分基于强度的差值，计算更快捷，而 SIFT 描述子通常在搜索正确的特征时更加精确。

参 考

前一则秘诀有关于 SURF 及 SIFT 特征更多的内容。

第 9 章

估算图像间的投影关系

本章主要探讨：

- ◆ 相机标定；
- ◆ 计算一对图像的基础矩阵（Fundamental Matrix）；
- ◆ 使用随机采样一致算法（RANSAC）进行图像匹配；
- ◆ 计算两幅图之间的单应矩阵（Homography）。

9.1 引言

图像通常由数码相机拍摄真实场景生成，将光线通过相机透镜投影到图像传感器上。图像是由 3D 场景投影到 2D 的平面上，因此场景与它的成像之间存在重要的联系，相同场景的不同成像之间也有联系。在数学术语中，投影几何学是描述成像过程的工具。本章将介绍多视角成像中基础的投影关系，以及在计算机视觉中如何使用它们。我们还会继续讨论前一章最后一则秘诀中提到的多视角特征匹配，将提出改进匹配结果的新策略。但是在这之前，让我们先介绍场景投影与图像成型中的基本概念。

成像

从根本上讲，自摄影技术诞生以来，用于成像的技术没有变过。来自场景的光是通过正面孔径被相机捕获，光线击中位于相机背面的图像平面（或图像传感器）。此外，镜头被用于集中来自不同场景元素的射线。图 9.1 说明了这个过程。

在这里， d_o 是观察对象到镜头的距离， d_i 是镜头到图像平面的距离， f 是镜头的焦距。这些量在薄透镜方程的关系如下：

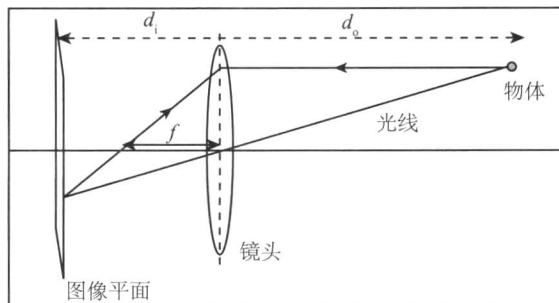


图 9.1

$$\frac{1}{f} = \frac{1}{d_o} + \frac{1}{d_i}$$

在计算机视觉中，这个相机模型可以在很多方面进行简化。首先，将摄像机的孔径视作无穷小，我们可以忽视透镜的影响，理论上这并不会改变成像。因此我们只考虑中心位置的射线。其次，由于大多数情况 d_o 远远大于 d_i ，我们可以假设图像平面位于焦距处。最后，我们可以从系统的几何结构中看到，平面上的图像是颠倒的。通过简单地将图像平面放置在镜头前，我们可以获得一个相同的笔直图像。这在物理上是不可行的，但从数学角度来看，完全是等价的。这个简化模型常常被称作针孔摄像机模型，它的表现形式如图 9.2 所示。

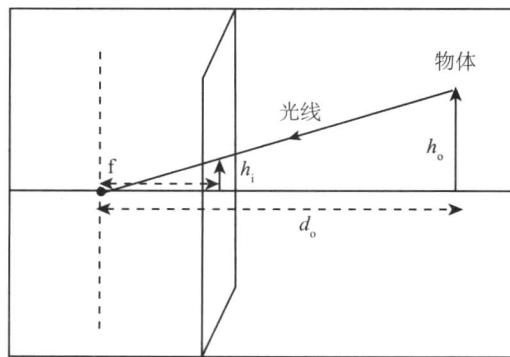


图 9.2

通过这个模型以及相似三角形的法则，我们可以轻易推导出基本的投影等式：

$$h_i = f \frac{h_o}{d_o}$$

物体（高度为 h_o ）在图像中的尺寸 (h_i) 因此与它离真实相机的距离 (d_o) 成反比，这个关系使得我们能够预测三维场景中点在图像中的位置。

9.2 相机标定

通过本节的介绍，我们将得知针孔照相机模型中的基本参数包括焦距和成像平面的尺寸（它定义了相机的视野）。同时，由于我们处理的是数字图像，成像平面中像素的数目是另一个重要特征。最后，为了能够计算场景点在图像中的像素坐标，我们还需要一个额外的信息。考虑到来自焦点的线与成像平面是正交的，我们需要知道这条线穿透图像平面的像素位置，这一点被称为主点。假设主点位于图像平面的中心在逻辑上是正确的，但在实践中由于相机生产时的不同精度它可能会偏移几个像素。

相机标定便是获取不同相机参数的过程，我们可以使用相机制造商提供的规格说明书，然而在特定应用如三维重建中，这些说明书的精确度是不够的。相机标定中，我们拍摄已知的图案并进行分析，之后的一个最优化过程中将确定满足观察的最优参数。这是个复杂的过程，但是 OpenCV 的标定函数降低了难度。

实现方法

为了标定相机，需要拍摄三维场景中坐标已知的一组。然后，必须确定这些点在图像中的位置。显然，为了得到准确的结果，我们需要观察多个点。一种方法是拍摄一幅由许多已知点组成的场景。为了更方便，我们也可以从不同角度多次拍摄一组点，这更简单但是需要计算每个视角的位置，同时还要计算相机的内参数，幸运的是这是可行的。

OpenCV 使用棋盘图案来生成一组三维点，每个点都位于方块的角点处。由于这是平面图案，我们可以假设棋盘位于 $Z=0$ 的平面上，而 X 轴和 Y 轴与网格对齐。标定的过程中，我们从不同视角展示棋盘图案，图 9.3 是标定图案的一个例子。

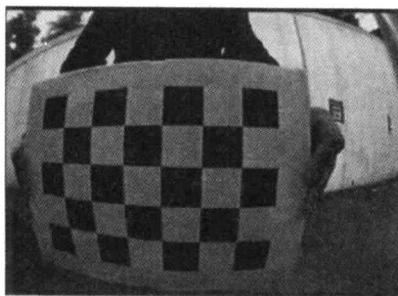


图 9.3

幸运的是，OpenCV 提供自动检测棋盘图案中角点的函数。你只需要提供一幅图像以及棋盘的尺寸（水平方向及垂直方向内部角点的个数）。该函数将返回棋盘角点在

图像中的位置，如果未能找到棋盘的图案，那么将返回 false：

```
// 输出图像角点的 vector  
std::vector<cv::Point2f>imageCorners;  
// 棋牌中角点数目  
cv::Size boardSize(6, 4);  
// 获取棋牌的角点  
bool found = cv::findChessboardCorners(image, boardSize, imageCorners);
```

如果需要对算法进行调优，那么该函数也接受额外的参数，但是我们不在此讨论。同时有用于在棋盘图像上绘制检测到角点的函数，通过直线依次将它们相连：

```
// 绘制角点  
cv::drawChessboardCorners(image,  
                           boardSize, imageCorners,  
                           found); // 已经找到的角点
```

获得的图像如图 9.4 所示。

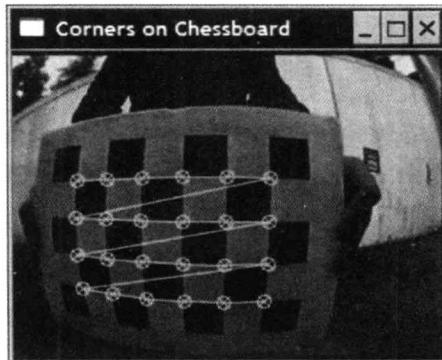


图 9.4

连接角点的直线展示了角点在向量中的顺序。为了标定相机，我们需要同时输入图像点以及对应的 3D 点。标定的过程被封装在 CameraCalibrator 类中：

```
class CameraCalibrator{  
    // 输入点：  
    // 位于世界坐标系的点  
    std::vector<std::vector<cv::Point3f>>objectPoints;  
    // 像素坐标系的点  
    std::vector<std::vector<cv::Point2f>>imagePoints;
```

```
// 输出矩阵
cv::Mat cameraMatrix;
cv::Mat distCoeffs;
// 标定的方式
int flag;
// 用于图像去畸变
cv::Mat map1, map2;
bool mustInitUndistort;
public:
    CameraCalibrator():flag(0),mustInitUndistort(true){};
```

如之前所说，只要我们将坐标系放置于棋盘上，那么角点的3D坐标可以轻易确定。实现这个过程的方法是以不同的棋盘图像文件名作为输入：

```
// 打开棋盘图像并提取角点
int CameraCalibrator::addChessboardPoints(
    const std::vector<std::string>& filelist,
    cv::Size & boardSize){
    // 棋盘上的点的两种坐标
    std::vector<cv::Point2f>imageCorners;
    std::vector<cv::Point3f>objectCorners;
    //3D 场景中的点：
    // 在棋盘坐标系中初始化棋盘角点
    // 这些点位于 (X,Y,Z)=(i,j,0)
    for(int i=0;i<boardSize.height;i++){
        for(int j=0;j<boardSize.width;j++){
            objectCorners.push_back(cv::Point3f(i,j,0.0f));
        }
    }
    //2D 图像中的点：
    cv::Mat image;// 为了保存棋盘图像
    int successes =0;
    //for all viewpoints
    for(int i = 0;i<filelist.size();i++){
        // 打开图像
        image = cv::imread(filelist[i],0);
        // 得到角点
        bool found = cv::findChessboardCorners(
```

```
    image, boardSize, imageCorners);  
    // 获取亚像素精度  
    cv::cornerSubPix(image, imageCorners,  
                      cv::Size(5,5),  
                      cv::Size(-1,-1),  
                      cv::TermCriteria(cv::TermCriteria::MAX_ITER+  
                                      cv::TermCriteria::EPS,  
                                      30,           // 最大迭代数目  
                                      0.1));       // 最小精度  
    // 如果角点数目满足要求，那么将它加入数据  
    if(imageCorners.size() == boardSize.area()) {  
        // 添加一个视角中的图像点及场景点  
        addPoints(imageCorners, objectCorners);  
        successes++;  
    }  
}  
return successes;  
}
```

第一个循环中输入棋盘的 3D 坐标，它们以任意的方形尺寸单位指定，而对应的图像点位于 `cv::findChessboardCorners` 的返回值中。每个可能的视角都会进行这个运算，然而为了获取更精确的像素坐标，我们还借助 `cv::cornerSubPix` 以得到如函数名所示的亚像素精度。`cv::TermCriteria` 对象是该函数迭代过程的终止条件，它定义了最大迭代数目以及亚像素坐标的最小精度，二者满足其一后迭代过程便终止。

成功检测到一组角点后，这些点被添加到坐标点的向量中：

```
// 添加场景点与对应的图像点  
void CameraCalibrator::addPoints(const std::vector<cv::Point2f>&  
imageCorners, const std::vector<cv::Point3f>& objectCorners) {  
    //2D 图像点  
    imagePoints.push_back(imageCorners);  
    // 对应 3D 场景中的点  
    objectPoints.push_back(objectCorners);  
}
```

该向量包含的是 `std::vector` 对象，其中每个向量元素都是取自某个视角的一组点。一旦我们处理了足够多的棋盘图像，并且获取了大量的 3D/2D 点对，那么我们可

以开始计算标定参数：

```
// 进行标定，返回重投影误差 (re-projection error)
double CameraCalibrator::calibrate(cv::Size& imageSize)
{
    // 必须重新进行去畸变
    mustInitUndistort = true;
    // 输出旋转和平移
    std::vector<cv::Mat> rvecs, tvecs;
    // 开始标定
    return
        calibrateCamera(objectPoints, // 3D 点
                        imagePoints,           // 图像点
                        imageSize,             // 图像尺寸
                        cameraMatrix,          // 输出的相机矩阵
                        distCoeffs,            // 输出的畸变矩阵
                        rvecs, tvecs,           // 旋转和平移
                        flag);                // 额外选项
}
```

实践中，10 ~ 20 幅棋盘图像便足够，但是它们必须是在不同距离不同视角进行拍摄。该函数的两个重要输出值是相机矩阵和畸变参数，相机矩阵将在之后描述。我们首先考虑畸变参数，我们介绍了针孔相机模型，但是忽略了透镜的影响。这只有在透镜不会带来明显光学畸变时才是可能的，不幸的是透镜的质量通常很差或者焦距过短。也许你已经注意到在我们所用的图像，棋盘图案明显发生了畸变，矩形板的边缘在如下中显示为曲线。我们也可以观察到图像的畸变离中心越远越明显。这是鱼眼透镜可以观察到的典型畸变，我们称之为径向畸变。常见数码相机中透镜不会发生如此明显的畸变，但是考虑到例子中使用的透镜，这种畸变显然是不可忽略的。

通过引入合适的模型，可以弥补这种变形，想法是通过一组数学公式表示这种畸变。一旦我们得到这公式，可以反转它以复原图像中的畸变。幸运的是，在相机标定过程中我们将获取畸变的参数以及其他相机参数。一旦这个步骤完成，之后图像中的畸变都可以去除：

```
// 标定后去除图像中的畸变
cv::Mat CameraCalibrator::remap(const cv::Mat &image) {
    cv::Mat undistorted;
    if(mustInitUndistort) { // 每次标定只需要初始化一次
```

```
cv::initUndistortRectifyMap(  
    cameraMatrix,           // 计算得到的相机矩阵  
    distCoeffs,            // 计算得到的畸变矩阵  
    cv::Mat(),             // 可选的 rectification 矩阵（此处为空）  
    cv::Mat(),             // 用于生成 undistorted 对象的相机矩阵  
    image.size(),          //undistorted 对象的尺寸  
    CV_32FC1,              // 输出的映射图像的类型  
    map1, map2);           //x 坐标和 y 坐标映射函数  
    mustInitUndistort = false;  
}  
  
// 应用映射函数  
cv::remap(image, undistorted, map1, map2,  
    cv::INTER_LINEAR); // 插值类型  
return undistorted;  
}
```

生成的图像如图 9.5 所示。

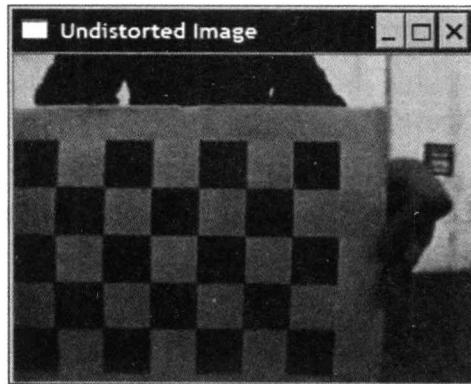


图 9.5

正如你将看到的，一旦图像的畸变被去除，我们便得到正规的透视图像。

作用原理

为了解释标定的结果，我们需要回到介绍中的图表，它描述了针孔相机模型。特别的是，我们希望阐述 3D 点 (X, Y, Z) 与 2D 像素点 (x, y) 之间的联系。让我们重新绘制图表并增加位于投影中心位置的坐标系，如图 9.6 所示。

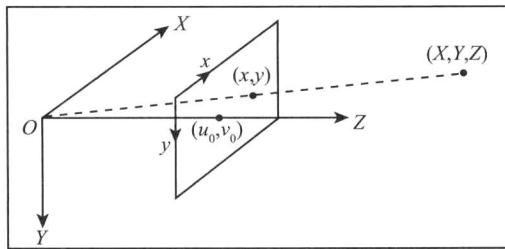


图 9.6

我们将 Y 轴竖直向下，得到的坐标系与图像原点位于左上角的惯例一致。我们之前了解到点 (X, Y, Z) 将投影到位于图像平面的 $(f_x/Z, f_y/Z)$ 位置上。如果需要将这个坐标转化为像素，需要将坐标分别除以像素宽度 (p_x) 以及高度 (p_y) 。通过除以世界单元（通常是米或毫米）的焦距 (f) ，我们得到水平像素表示的焦距，定义它为 f_x 。类似的， $f_y = f/p_y$ 用于定义垂直像素表示的焦距。完整的投影等式如下：

$$x = \frac{f_x X}{Z} + u_0$$

$$y = \frac{f_y Y}{Z} + v_0$$

主点 (u_0, v_0) 被加在结果上，将原点移动到图像的左上角。这个等式可以以矩阵的形式重写，为此我们引入齐次坐标的概念，用三维向量来表示 2D 点，用四维向量来表示 3D 点。额外的一个坐标是任意的缩放因子，从三维向量中提取 2D 坐标时需要去除它的影响。

$$s \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

第二个矩阵是投影矩阵，第一个 3×3 矩阵被称为相机内参数矩阵，它是 `cv::calibrateCamera` 函数的输出矩阵。同时还有一个函数 `cv::calibrationMatrixValues` 输入标定矩阵后返回内参数的值。

更一般的是，当参考坐标系并不位于相机的投影中心，我们需要添加一个旋转向量 (3×3 矩阵) 以及一个平移向量 (3×1 矩阵)，这两个矩阵描述的刚体 (Rigid) 变换将使得 3D 点回到相机参考坐标系。因此，我们可以通过最一般的形式重写投影等式：

$$s \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

在我们的标定例子中，参考坐标系位于棋盘上，因此每个视角都必须计算相应的刚体变换（旋转及平移），它们位于 `cv::calibrateCamera` 函数的输出参数列表中。旋转和平移分量通常被称为标定的外参数，在不同的视角中是不同的。对于给定的相机和透镜系统，内参数总是保持不变的。在我们的例子中，基于 20 幅棋盘图像标定得到的内参数为： $f_x=167$, $f_y=178$, $u_0=156$, $v_0=119$ 。在寻找这些结果时，`cv::calibrateCamera` 使用最优化过程期望找到最优参数，使得 3D 场景中的点在图像上的预测投影点与真实图像中的点之间的差值最小，点的差值之和被称为重投影误差。

为了校正畸变，OpenCV 使用多项式函数移动图像点到它们正确的位置上。默认情况下使用 5 个系数，也可以使用 8 个参数。一旦我们获得这些参数，可以计算两个映射函数（分别是 x 坐标以及 y 坐标），它们将给出新的没有畸变的图像坐标点。由于这是非线性变换，一些像素将落在输出图像的边界之外。通过扩大输出图像的尺寸可以弥补这个像素的损失，但是将得到原图像中不存在的像素（可以显示为黑色）。

扩展阅读

当我们实现拥有相机内参数的良好估计值，可以使用它作为 `cv::calibrateCamera` 的参数，它将用作最优化过程的初始值，当然你还需要增加标记位 `CV_CALIB_USE_INTRINSIC_GUESS`。也可以指定 `CV_CALIB_FIX_PRINCIPAL_POINT` 以使用固定的主点值，因为通常它位于中心点。如果假设像素是方形的，那么可以指定 `CV_CALIB_FIX_RATIO` 以使用固定的焦距 f_x 及 f_y 之间的比率。

9.3 计算一对图像的基础矩阵

之前的秘诀中我们展示了如何恢复单个相机的投影等式。本则秘诀将探索两幅观察相同场景的图像之间的投影关系，这两幅图像既可以是同一个相机在不同位置拍摄的，也可以是两个相机拍摄的。当这两个相机被刚体基线（Rigid Baseline）分隔，那么我们使用术语立体视觉（Stereovision）。

准备工作

首先考虑观察场景点的两个相机如图 9.7 所示。

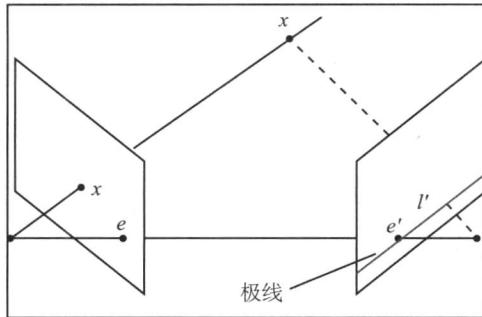


图 9.7

我们可以找到 3D 点 X 的图像点 x , 只需追踪连接 3D 点与相机中心的直线。反过来, 位于 x 位置的图像点可以位于 3D 空间中该直线中任意点上。这意味着如果我们需要通过图像点找到另一幅图像中的对应点, 必须沿着这条直线在另一个图像平面进行搜索。这条虚构的直线被称为点 x 的极线 (Epipolar), 它定义了两个对应点必须满足的基础限制, 即它们必须位于彼此的极线上。极线的精确朝向依赖于两个相机的相对位置, 事实上极线的位置描述了双视角系统的几何结构。

从这个双视角系统的几何结构中我们还观察到, 所有极线都穿过同一个点, 该点对应于其中一个相机的中心在另一个相机上的投影, 我们称之为极点 (Epipole) :

使用数学术语, 图像上点与它对应的极线之间的关系可使用 3×3 矩阵表示:

$$\begin{bmatrix} l'_1 \\ l'_2 \\ l'_3 \end{bmatrix} = F \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

在投影几何中, 2D 直线也可以表示为三维向量, 它对应于满足等式 $l'_1x + l'_2y + l'_3 = 0$ 的 2D 点的集合 (x', y') (位于上方的标记意味着该直线属于第二幅图像)。因此, 基础矩阵 F 将一个视角中的 2D 图像点映射到另一个视角中的极线上。

实现方法

估算一对图像的基础矩阵可以通过求解一组等式, 其中涉及两幅图像之间的已知匹

配点，最小的匹配数量是 7。使用前一章的图像对，我们可以手动选出 7 个高质量的匹配（可以在图 9.8 中看到）。cv::findFundamentalMat 函数将使用它们计算基础矩阵。

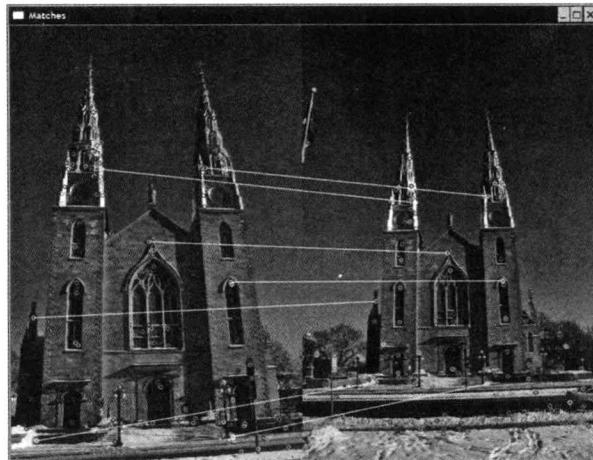


图 9.8

如果我们以 cv::KeyPoint 对象的形式获得图像点的信息，它们必须先转化为 cv::Point2f，因为 cv::findFundamentalMat 只接受后者。用于进行转换的函数为

```
// 转换 KeyPoint 类型到 Point2f
std::vector<cv::Point2f> selPoints1, selPoints2;
cv::KeyPoint::convert(keypoints1, selPoints1, pointIndexes1);
cv::KeyPoint::convert(keypoints2, selPoints2, pointIndexes2);
```

向量 selPoints1 及 selPoints2 包含着两幅图像中的对应点，keypoints1 及 keypoints2 则是如前一章所示的被选中的特征点。调用 cv::findFundamentalMat 函数的方法如下：

```
// 从 7 个匹配中计算 F 矩阵
cv::Mat fundamental = cv::findFundamentalMat(
    cv::Mat(selPoints1), // 图 1 中的点
    cv::Mat(selPoints2), // 图 2 中的点
    CV_FM_7POINT); // 使用 7 个点的方法
```

一种在视觉上验证基础矩阵是否有效的方法是对选中的点绘制极线。另一个 OpenCV 函数允许指定一组点，计算它们的极线。一旦计算完成，可以通过 cv::line 函数进行绘制。下述代码计算左图中点的极线，并且绘制在右图中：

```

// 在右图中绘制对应的极线
std::vector<cv::Vec3f> lines1;
cv::computeCorrespondEpilines(
    cv::Mat(selPoints1), // 图像点
    1,                  // 图 1 (也可以是 2)
    fundamental,        // F 矩阵
    lines1);           // 一组极线
// 对于所有极线
for(vector<cv::Vec3f>::const_iterator it= lines1.begin();
    it!=lines1.end();++it){
    // 绘制第一列与最后一列之间的直线
    cv::line(image2,
              cv::Point(0,-(*it)[2]/(*it)[1]),
              cv::Point(image2.cols,-((*it)[2]+
                                     (*it)[0]*image2.cols)/(*it)[1]),
              cv::Scalar(255,255,255));
}

```

结果如图 9.9 所示。

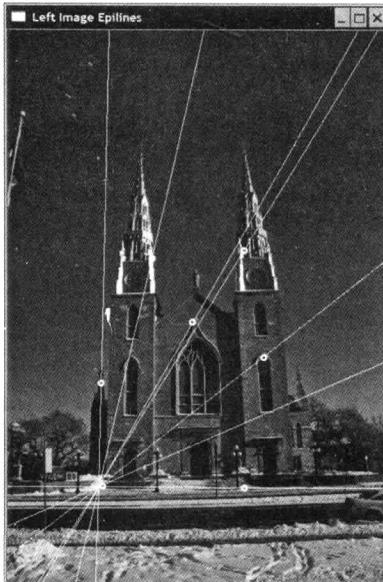


图 9.9

之前提到极点位于所有极线的交点处，并且是另一个相机的中心的投影。在先前的图像中极点是可见的。通常，极线相交于图像的边界之外，它位于当两幅图像同时拍摄时第一个相机可见的位置。观察这对图像，花些时间从中找到感觉。

作用原理

我们之前解释到对于图像中的一个点，基础矩阵给出一个直线方程，这个点在另一个视角中的对应点可以在直线上找到。如果(用齐次坐标表示的)点 p 的对应点是 p' ，同时 F 是两个视角之间的基础矩阵，那么由于 p' 位于极线 Fp 上，我们可以得到：

$$p'^T F p = 0$$

该方程表示了两个对应点之间的联系，被称为极性约束。使用该方程可以使用已知的配对估算矩阵的内容。由于 F 矩阵中每项都会受到缩放影响，因此仅有8个项需要估算（第9项可以设置为1）。每个匹配带来一个方程式，因此在已知8个匹配的情况下，我们可以通过求解线性方程组完整地计算出矩阵。这正是设置CV_FM_8POINT参数时cv:::findFundamentalMat函数的功能。需要注意的是，我们输入的匹配数目可以大于8（也建议如此），得到的线性方程的超定系统可以通过最小二乘法求解。

为了估算基础矩阵，有时会利用额外的约束条件。在数学上， F 矩阵将2D点映射到相交于一点的直线上。所有极线都通过这个特定的极点意味着矩阵的一个约束条件。该约束将所需的匹配数目降低到7。不幸的是，在这种情况下，方程组变成拥有3个可能解的非线性系统。在OpenCV中，给定CV_FM_7POINT标志位可以使用7个匹配进行基础矩阵计算。这也是我们在之前的例子中所做的。

最后，我们需要了解到在图像中选择合适的匹配对于结果的精确度是非常重要的。通常，匹配应该在图像间均匀分布，并且包含场景中不同深度的点。否则，将得到不稳定的或是退化后的结果。

参 考

由R.Hartley及A.Zisserman撰写的“*Multiple View Geometry in Computer Vision*”(Cambridge University Press, 2004)是计算机视觉中投影几何学最完整的参考资料。

下一节将展示OpenCV计算基础矩阵时可用的一个额外标志位。

9.4 使用随机采样一致算法(RANSAC)进行图像匹配

当两个相机观察的是同一个场景，它们能看到是不同视角中的相同元素。我们在前一章中研究过特征点匹配的问题，在本则秘诀中，我们将回到这个问题，并且学习如何利用双视角的极性约束来更加可靠地匹配图像特征。

原理很简单：当我们匹配两幅图像间的特征点时，仅仅接受落在对应极线中的匹配对。然而，为了能够验证该条件，基础矩阵必须是已知的，同时用于估算该矩阵的匹配必须是高质量的。这似乎就成了先有鸡还是先有蛋的问题。本则秘诀中将提出的解决方案将同时计算基础矩阵与优质匹配集合。

实现方法

我们的目标是能够获取双视角间的优质匹配集合。因此，所有检测到的特征匹配将使用前一则秘诀中介绍的极性约束进行验证。我们首先定义一个封装类：

```
class RobustMatcher{
private:
    // 指向特征检测器的智能指针
    cv::Ptr<cv::FeatureDetector>detector;
    // 指向描述子提取器的智能指针
    cv::Ptr<cv::DescriptorExtractor>extractor;
    float ratio;// 第一个以及第二个最近邻之间的最大比率
    bool refineF;// 是否改善 F 矩阵
    double distance;// 到极线的最小距离
    double confidence;// 置信等级（概率）
public:
    RobustMatcher():ratio(0.65f),refineF(true),
                    confidence(0.99),distance(3.0){
        //SURF 为默认特征
        detector= new cv::SurfFeatureDetector();
        extractor= new cv::SurfDescriptorExtractor();
    }
}
```

我们使用通用的 `cv::FeatureDetector` 及 `cv::DescriptorExtractor` 接口，因此用户能够使用选择不同的实现。这里默认使用 SURF 特征以及描述子，但是也可以通过适当地设置函数指定其他的特征及描述子：

```
// 设置特征检测器
void setFeatureDetector(
cv::Ptr<cv::FeatureDetector>& detect){
    detector=detect;
}
// 设置描述子提取器
void setDescriptorExtractor(
cv::Ptr<cv::DescriptorExtractor>& desc){
```

```
    extractor= desc;
}
```

match 方法是最核心的方法，它返回匹配集合、检测到的特征点以及估算的基础矩阵。我们依次进行五个不同操作，代码的注释会明确标示这些步骤：

```
// 使用对称性测试以及 RANSAC 匹配特征点
// 返回基础矩阵
cv::Mat match(cv::Mat& image1,
              cv::Mat& image2, // 输入图像
              // 输出匹配及特征点
              std::vector<cv::DMatch>& matches,
              std::vector<cv::KeyPoint>& keypoints1,
              std::vector<cv::KeyPoint>& keypoints2) {
    //1a. 测 SURF 特征
    detector->detect(image1, keypoints1);
    detector->detect(image2, keypoints2);
    //1b. 取 SURF 描述子
    cv::Mat descriptors1, descriptors2;
    extractor->compute(image1, keypoints1, descriptors1);
    extractor->compute(image2, keypoints2, descriptors2);
    //2. 配两幅图像的描述子
    // 创建匹配器
    cv::BruteForceMatcher<cv::L2<float>>matcher;
    // 从图 1-> 图 2 的 k 最近邻 (k=2)
    std::vector<std::vector<cv::DMatch>>matches1;
    matcher.knnMatch(descriptors1, descriptors2,
                      matches1, // 匹配结果的向量 (每项有两个值)
                      2);           // 返回两个最近邻
    // 从图 2-> 图 1 的 k 个最近邻 (k=2)
    std::vector<std::vector<cv::DMatch>>matches2;
    matcher.knnMatch(descriptors2, descriptors1,
                      matches2, // 匹配结果的向量 (每项有两个值)
                      2);           // 返回两个最近邻
    //3. 移除 NN 比率大于阈值的匹配
    // 清理图 1-> 图 2 匹配
    int removed= ratioTest(matches1);
    // 清理图 2-> 图 1 的匹配
    removed= ratioTest(matches2);
    //4. 移除非对称的匹配
    std::vector<cv::DMatch>symMatches;
    symmetryTest(matches1, matches2, symMatches);
```

```
//5. 使用 RANSAC 进行最终验证
cv::Mat fundamental= ransacTest(symMatches,
    keypoints1,keypoints2,matches);
// 返回找到的基础矩阵
return fundamental;
}
```

第一步仅仅是检测特征点并计算它们的描述子。接着，我们使用 `cv::BruteForceMatcher` 类进行特征匹配，这与前一章所做的没什么不同。然而，这次对于每个特征都搜寻两最匹配的特征点（之前我们只搜索一个点），这通过调用 `cv::BruteForceMatcher::knnMatch` 方法实现（参数 $k=2$ ）。此外，我们进行双向的匹配，即找到图 1 中每个特征点在图 2 中的两个最佳匹配，然后找到图 2 中每个特征点在图 1 中的两个最佳匹配。

因此，对于每个特征点，在另一个视角中都有两个候选的匹配点，它们是基于描述子间距离的两个值。如果对于最优值的测量距离非常低，而次优值的要大许多，那么我们可以安全地接受最优值，因为它毫无歧义地是最佳选择。反过来，如果两个候选者非常接近，那么选择其中之一可能出错，因此这两个匹配值都会被拒绝。在第三步中我们判断最优值与次优值之间距离的比率，看它是否不大于给定的阈值：

```
// 移除 NN 比率大于阈值的匹配
// 返回移除点的数目
// (对应的项被清零，即尺寸将为 0)
int ratioTest(std::vector<std::vector<cv::DMatch>>&matches) {
int removed=0;
// 遍历所有匹配
for(std::vector<std::vector<cv::DMatch>>::iterator
    matchIterator= matches.begin();
    matchIterator!= matches.end();++matchIterator){
    // 如果识别两个最近邻
    if(matchIterator->size()>1){
        // 检查距离比率
        if(((*matchIterator)[0].distance/
            (*matchIterator)[1].distance)>ratio) {
            matchIterator->clear();// 移除匹配
            removed++;
        }
    }else{// 不包含两个最近邻
        matchIterator->clear();// 移除匹配
        removed++;
    }
}
```

```

    }
}

return removed;
}

```

大量的引起歧义的匹配将在这个过程中被移除，这点可以在下例中看到。通过一个较低的 SURF 阈值 (=10)，我们最初检测到 1600 个特征点（黑色圆），其中仅有 55 个通过比率测试（白色圆），如图 9.10 所示。

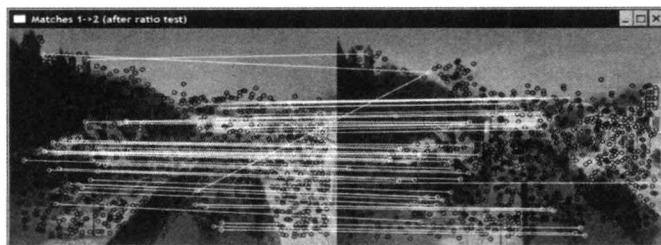


图 9.10

连接匹配点的白线表明即便我们获得了许多正确的匹配，还是有明显数目的错误匹配通过了测试。因此，我们执行第二次测试以过滤掉更多的错误匹配。对于第二次的匹配测试我们也使用比率测试。

现在我们有两个相对优质的匹配集，一个来自图 1 到图 2，另一个来自图 2 到图 1。我们将提取出同时满足两个集合的匹配，这是对称性匹配策略的特点，为了接受一对特征点，它们必须是各自的最优匹配点：

```

// 在 symMatches 向量中插入对称匹配
void symmetryTest(
    const std::vector<std::vector<cv::DMatch>>& matches1,
    const std::vector<std::vector<cv::DMatch>>& matches2,
    std::vector<cv::DMatch>& symMatches) {
    // 遍历图 1-> 图 2 的所有匹配
    for(std::vector<std::vector<cv::DMatch>>::iterator
        const_iterator matchIterator1=matches1.begin();
        matchIterator1!= matches1.end();++matchIterator1){
        // 忽略被删除的匹配
        if(matchIterator1->size()<2)
            continue;
        // 遍历图 2-> 图 1 的所有匹配
        for(std::vector<std::vector<cv::DMatch>>::iterator
            const_iterator matchIterator2=matches2.begin();

```

```
matchIterator2!= matches2.end();
++matchIterator2);
// 忽略被删除的匹配
if(matchIterator2->size()<2)
    continue;
// 对称性测试
if((*matchIterator1)[0].queryIdx ==
    (*matchIterator2)[0].trainIdx &&
    (*matchIterator2)[0].queryIdx ==
    (*matchIterator1)[0].trainIdx) {
    // 添加对称的匹配
    symMatches.push_back(
        cv::DMatch((*matchIterator1)[0].queryIdx,
                   (*matchIterator1)[0].trainIdx,
                   (*matchIterator1)[0].distance));
    break; // 图 1-> 图 2 中的下一个匹配
}
}
}
}
```

在我们的测试对中，31 个匹配通过这次对称性测试。最后的测试由一个额外的滤波试验组成，将使用基础矩阵来移除不满足极性约束的匹配。该测试基于 RANSAC 方法，即便匹配集合中存在异常值，它也可以计算基础矩阵：

```
// 基于 RANSAC 识别优质匹配
// 返回基础矩阵
cv::Mat ransacTest(
    const std::vector<cv::DMatch>& matches,
    const std::vector<cv::KeyPoint>& keypoints1,
    const std::vector<cv::KeyPoint>& keypoints2,
    std::vector<cv::DMatch>& outMatches) {
    // 转换 KeyPoint 类型到 Point2f
    std::vector<cv::Point2f> points1, points2;
    for(std::vector<cv::DMatch>::
        const_iterator it= matches.begin();
        it!= matches.end(); ++it) {
        // 得到左边特征点的坐标
        float x= keypoints1[it->queryIdx].pt.x;
        float y= keypoints1[it->queryIdx].pt.y;
        points1.push_back(cv::Point2f(x,y));
    }
}
```

```
// 得到右边特征点的坐标
x= keypoints2[it->trainIdx].pt.x;
y= keypoints2[it->trainIdx].pt.y;
points2.push_back(cv::Point2f(x,y));
}

// 基于 RANSAC 计算 F 矩阵
std::vector<uchar> inliers(points1.size(),0);
cv::Mat fundamental= cv::findFundamentalMat(
    cv::Mat(points1),cv::Mat(points2), // 匹配点
    inliers, // 匹配状态 (inlier 或 outlier)
    CV_FM_RANSAC, // RANSAC 方法
    distance, // 到极线的距离
    confidence); // 置信概率

// 提取通过的匹配
std::vector<uchar>::const_iterator
itIn= inliers.begin();
std::vector<cv::DMatch>::const_iterator
itM= matches.begin();

// 遍历所有匹配
for(;itIn!= inliers.end();++itIn,++itM){
    if(*itIn){ // 为有效匹配
        outMatches.push_back(*itM);
    }
}
if(refineF){
    // F 矩阵将使用所有接受的匹配重新计算
    // 转换 KeyPoint 类型到 Point2f
    // 准备计算最终的 F 矩阵
    points1.clear();
    points2.clear();
    for(std::vector<cv::DMatch>::
        const_iterator it= outMatches.begin();
        it!= outMatches.end();++it){
        // 得到左边特征点的坐标
        float x= keypoints1[it->queryIdx].pt.x;
        float y= keypoints1[it->queryIdx].pt.y;
        points1.push_back(cv::Point2f(x,y));
        // 得到右边特征点的坐标
        x= keypoints2[it->trainIdx].pt.x;
        y= keypoints2[it->trainIdx].pt.y;
        points2.push_back(cv::Point2f(x,y));
    }
}
```

```
// 从所有接受的匹配中计算 8 点 F
fundamental= cv::findFundamentalMat(
    cv::Mat(points1),cv::Mat(points2), // 匹配
    CV_FM_8POINT); //8 点法
}
return fundamental;
}
```

这段代码有点长，因为特征点在计算 F 矩阵前需要转换为 `cv::Point2f`。
完整的匹配过程使用 `RobustMatcher` 类，方法如下：

```
// 准备匹配器
RobustMatcher rmatcher;
rmatcher.setConfidenceLevel(0.98);
rmatcher.setMinDistanceToEpicenter(1.0);
rmatcher.setRatio(0.65f);
cv::Ptr<cv::FeatureDetector>pfd=
    new cv::SurfFeatureDetector(10);
rmatcher.setFeatureDetector(pfd);
// 匹配两幅图像
std::vector<cv::DMatch>matches;
std::vector<cv::KeyPoint>keypoints1,keypoints2;
cv::Mat fundamental= rmatcher.match(image1,image2,
    matches,keypoints1,keypoints2);
```

如图 9.11 所示，我们得到 23 个匹配结果，图中画有对应的极线。



图 9.11

作用原理

在之前的秘诀中，我们学习到能够从一组特征点的匹配中估算出与两幅图像相关的基础矩阵。精确地说，这组匹配仅包含高质量的匹配。然而，在实际应用中，很难保证通过比较特征点描述子得到的匹配集是完全相同的。因此我们将使用 RANSAC (RANdom SAMpling Consensus) 策略来进行基础矩阵的估算。

RANSAC 算法的目的是从包含异常值的数据集中估算出给定的数学元素。基本原理是随机地选取一些数据点，并且仅用它们来进行估算。选择的数据点的个数应当是可用于进行估算的最小数。对于基础矩阵而言，8 个匹配是最小数（事实上可以是 7，但是 8 个点的线性算法计算更迅速）。一旦从这随机的 8 个匹配中算出基础矩阵，集合中所有剩下的匹配都将与矩阵对应的极性约束进行测试。我们找到所有满足该约束的匹配，它们对应的特征非常靠近极线。这些匹配组成了这个基础矩阵的支持集合。

RANSAC 算法背后最主要的想法是支持集合越大，得到正确矩阵的可能性就越大。显而易见的，如果一个（或多个）随机选择的匹配是错误的，那么得到的基础矩阵也是错误的，于是它的支持集合应当很小。这个过程会重复数次，最后我们保留最大支持集合的矩阵作为最合适的。

因此，我们的目标是多次随机挑选 8 个匹配，最终能够得到 8 个足够好的匹配提供我们一个较大支持集合。根据完整数据集中错误匹配的个数，挑选到 8 个正确数据的概率会不同。然而我们值得挑选的次数越多，我们从中得到至少一个优质匹配集合的概率就越大。更精确地，如果假设集合包含 $n\%$ 正确值，那么同时选中 8 个正确匹配的概率是 $8n$ 。因此，包含至少一个错误匹配的概率是 $(1-8n)$ 。如果我们挑选 k 次，至少出现一次包含 8 个正确结果的概率是 $1 - (1-8n)^k$ 。这便是置信概率 c ，我们希望它尽可能地高。因此，当运行 RANSAC 算法时，我们需要确定 k 的数量以得到给定的置信等级。

在 `cv::findFundamentalMat` 函数中使用 RANSAC 算法时，需要提供两个额外的参数。第一个是置信等级，它决定迭代的次数。第二个是归类为正确的点离极线的最大距离。因此，该函数返回一个字符类型的 `std::vector`，标志着对应的匹配被识别为 `outlier(0)` 还是 `inlier(1)`。

初始数据集中优质匹配的数量越大，RANSAC 给出正确基础矩阵的概率也越高。因此我们在调用 `cv::findFundamentalMat` 之前便对数据集使用了多个滤波器。当然，你可以选择跳过其中的一些步骤。这仅仅是如何在计算复杂性、最终匹配数目以及所需的置信等级之间进行平衡的问题。

9.5 计算两幅图之间的单应矩阵

本章的 9.2 节展示了如何由一组匹配计算基础矩阵，我们还能从匹配对中计算另一个数学元素：单应矩阵（Homography）。和基础矩阵相似，单应矩阵是一个 3×3 的矩阵，它有着特殊的属性可以用于特定条件下的双视角图像。

准备工作

本章 9.1 节介绍过，3D 点与它在相机图像中像素点之间存在投影关系，基本上我们知道这个关系可以用 3×4 矩阵表示。现在，如果考虑一个特殊的案例，即同一场景的两个视图的区别仅仅是一个纯粹的旋转，那么可以发现这个外部矩阵（Extrinsic Matrix）的第 4 列都是由 0 组成的（即不存在平移）。于是，投影关系变成了 3×3 矩阵，即单应矩阵。它意味着，在特殊的条件下（这里是一个纯粹的旋转），同一个点在不同视图中的图像存在线性关系：

$$\begin{bmatrix} sx' \\ sy' \\ s \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

在齐次坐标系中，这个关系中包含缩放因子，即标量 s 。一旦我们估算出这个矩阵的值，那么使用这个关系一个视图中所有的点都可以变换到另一个视图中。作为纯粹旋转的单应关系的副作用，这里基础矩阵的值是未定义的。

实现方法

假设我们的两幅图像仅用旋转便可以区分，它们可以使用 `RobustMatcher` 进行匹配，只是我们跳过了 RANSAC 验证的步骤（匹配函数中的第 5 步），因为它用于估算基础矩阵。我们将增加一个 RANSAC 步骤，它用于从包含大量错误匹配的数据集中估算单应矩阵。使用的是 `cv::findHomography` 函数，它与 `cv::findFundamentalMat` 非常类似：

```
// 找到两个图之间的单应矩阵
std::vector<uchar> inliers(points1.size(), 0);
cv::Mat homography = cv::findHomography(
```

```
cv::Mat points1, // 对应的  
cv::Mat points2, // 点集  
inliers,          // 输出的正确值  
CV_RANSAC,        // RANSAC 算法  
1.);             // 到反投影点的最大距离
```

单应矩阵仅仅在两幅图像可以通过纯粹的旋转区分时才存在，这正是图 9.12 中两幅图像的情况。

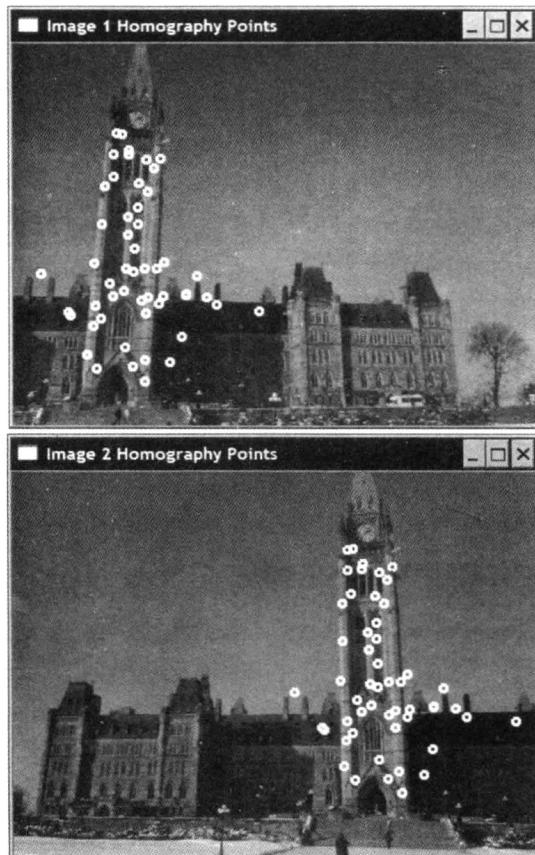


图 9.12

得到的符合单应矩阵的正确值在下面的循环中绘制到那些图像上：

```
// 绘制 inlier 点  
std::vector<cv::Point2f>::const_iterator itPts=points1.begin();
```

```
std::vector<uchar>::const_iterator itIn= inliers.begin();
while(itPts!=points1.end()){
    // 在每个 inlier 位置画圆
    if(*itIn)
        cv::circle(image1,*itPts,3,
                   cv::Scalar(255,255,255),2);
    ++itPts;
    ++itIn;
}
itPts=points2.begin();
itIn=inliers.begin();
while(itPts!=points2.end()){
    // 在每个 inlier 位置画圆
    if(*itIn)
        cv::circle(image2,*itPts,3,
                   cv::Scalar(255,255,255),2);
    ++itPts;
    ++itIn;
}
```

一旦我们得到单应矩阵，可以将像素点从一幅图像变换到另一幅。事实上，你可以对图像中所有的像素这么做，得到的结果将是这幅图像被变换到了另一个视角。负责该功能的 OpenCV 函数是：

```
// 歪曲 image 1 到 image 2
cv::Mat result;
cv::warpPerspective(image1,// 输入
                    result,           // 输出
                    homography,       //homography
                    cv::Size(2*image1.cols,
                            image1.rows));// 输出图像的尺寸
```

一旦获得新的图像，可以添加到另一幅图像上以扩展视角（因为两幅图像现在有着相同的视角）：

```
// 赋值图 1 到整幅图像的前半部分
cv::Mat half(result,cv::Rect(0,0,image2.cols,image2.rows));
image2.copyTo(half); // 复制图 2 到图 1 的 ROI 区域
```

得到的图像如图 9.13 所示。

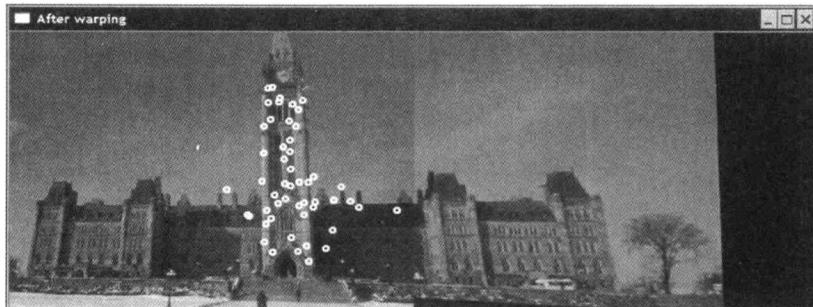


图 9.13

作用原理

当两个视角通过一个单应矩阵关联，可以确定给定的场景点在另一幅图像上的位置。这个特性对于落在图像边界外的点非常有趣。事实上，由于视角 2 展示了图 1 中不可见的部分场景，你可以使用单应矩阵来将图 2 中的额外像素扩展到图 1 中，额外的像素列被添加到图像的右侧。

由 `cv::findHomography` 计算而得的单应矩阵将图 1 中的点映射到图 2 中。事实上，我们需要用来变换图 1 中点到图 2 的是单应矩阵的逆矩阵。这正是函数 `cv::warpPerspective` 默认实现的功能，得到的是输出中每个点的颜色值。当输出像素变换到输入图像之外时，该像素被赋予黑色值（0）。可选的标志位 `cv::WARP_INVERSE_MAP` 可以作为 `cv::warpPerspective` 的第 5 个参数，它表明在像素变换时直接使用单应矩阵，不适用逆矩阵。

扩展阅读

单应矩阵也存在于一个平面的两个视角中，通过再次查看相机投影等式便可知道。当观察的是平面时，我们可以设置参考坐标系使得平面上所有点的 Z 坐标都是 0。这也会消去 3×4 投影矩阵的某一行，生成 3×3 矩阵，即单应矩阵。这意味着，如果你拥有建筑平坦外表面的不同视角的照片，那么可以计算这些图像间的单应矩阵，并且将图像拼接成一幅大的图像。

两两视角之间最少需要两个匹配点便可以计算一个单应矩阵。函数 `cv::getPerspectiveTransform` 可以从两个点构建这么一个变换矩阵。

第 10 章

处理视频序列

本章主要探讨：

- ◆ 读取视频序列；
- ◆ 处理视频帧；
- ◆ 写入视频序列；
- ◆ 跟踪视频中的特征点；
- ◆ 提取视频中的前景物体。

10.1 引言

视频信号是视觉信息的一大源头，它由按序列排放的图像组成，即帧（Frame）。这些图像通过有规律的间隔进行拍摄（以帧率的形式指定），它们展示了动态的场景。利用性能强劲的计算机，我们能够对视频序列进行视觉分析，有时能接近、甚至快于真实的视频帧率。本章将展示如何读取、处理以及存储视频序列。

一旦我们能提取视频中的每个单独帧，那么书中展示过的不同图像处理函数可以应用到每个帧上。此外，我们也将介绍多个对视频序列实施时序分析的算法，比较相邻帧以进行物体跟踪，随时间累积图像统计数据以提取前景物体。

10.2 读取视频序列

为了处理视频序列，我们需要读取每一帧。OpenCV 提供了一个易用的框架以提取视频文件和 USB 摄像头中的图像帧。本则秘诀将展示如何使用它。

实现方法

基本上，你只需要创建一个 `cv::VideoCapture` 实例，然后在循环中提取并读

取每一帧。这个基本的 main 函数仅仅是显示视频序列的每一帧：

```
int main()
{
    // 打开视频文件
    cv::VideoCapture capture("../bike.avi");
    // 检查视频是否成功打开
    if(!capture.isOpened())
        return 1;
    // 获取帧率
    double rate= capture.get(CV_CAP_PROP_FPS);
    bool stop(false);
    cv::Mat frame;// 当前视频帧
    cv::namedWindow("Extracted Frame");
    // 每一帧之间的延迟
    // 与视频的帧率相对应
    int delay= 1000/rate;
    // 遍历每一帧
    while(!stop){
        // 尝试读取下一帧
        if(!capture.read(frame))
            break;
        cv::imshow("Extracted Frame",frame);
        // 引入延迟
        // 也可通过按键停止
        if(cv::waitKey(delay)>=0)
            stop=true;
    }
    // 关闭视频文件
    // 将由析构函数调用，因此非必须
    capture.release();
}
```

将出现一个窗口，如图 10.1 所示播放视频。

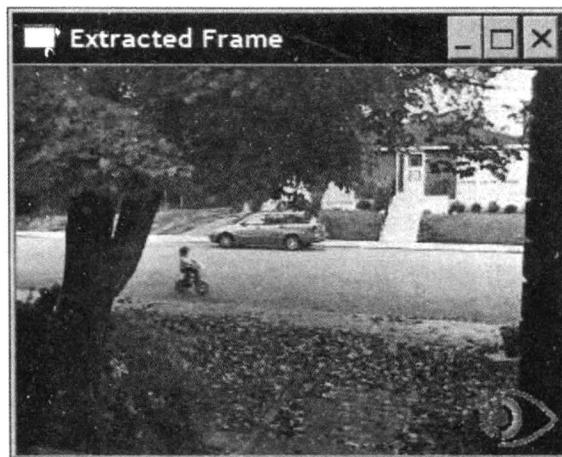


图 10.1

作用原理

为了播放视频，你只需要在创建 `cv::VideoCapture` 对象时指定视频的文件名，你也可以在创建 `cv::VideoCapture` 之后再调用 `open` 函数。一旦视频成功打开（可通过 `isOpened` 函数进行验证），便可以开始提取图像帧。也可以通过 `get` 方法查询不同的视频信息。在上例中，我们通过 `CV_CAP_PROP_FPS` 标志位获得视频的帧率。由于它是一个通用的函数，总是返回一个 `double` 值，即便我们期望获取的是其他类型的值。例如，视频文件的帧数是一个整数，我们可以这么获取：

```
long t= static_cast<long>(
    capture.get(CV_CAP_PROP_FRAME_COUNT));
```

阅读 OpenCV 文档可以了解到能够从视频中获取到的不同信息。

相对应地，也有一个 `set` 函数用于设置 `cv::VideoCapture` 对象的参数。例如，通过 `CV_CAP_PROP_POS_FRAMES` 你可以移动视频到特定的帧：

```
// 跳至帧 100
double position= 100.0;
capture.set(CV_CAP_PROP_POS_FRAMES,position);
```

你也可以使用 `CV_CAP_PROP_POS_MSEC` 以毫秒的方式指定播放位置，或通过 `CV_CAP_PROP_POS_AVI_RATIO` 指定相对位置，0.0 对应的是视频的开始，1.0 则

是结尾。该参数设置成功时，该方法返回 `true`。是否能够获取和设置某个视频参数取决于视频的编码方式，如果你失败，很可能只是因为选用的编码不支持该参数。

一旦视频成功打开，可以通过重复调用 `read` 方法获取视频序列，这正是上例所做的。也可以调用重载过的操作符：

```
capture>>frame;
```

另外，也可以调用两个基本函数：

```
capture.grab();
capture.retrieve(*frame);
```

在我们的例子中，`cv::waitKey` 函数在显示每一帧之间引入了延迟。我这里设置的延迟对应于输入视频的帧率，如果 `fps` 是每秒的帧数，那么 $1000/fps$ 毫秒便是两帧之间的延迟。你可以改变该值，以更快或更慢的速率播放视频。窗口的优先级比较低，如果 CPU 过于繁忙，它将不会刷新内容。因此如果你希望窗口有足够的时间刷新，那么插入这样一个延迟是必需的。`cv::waitKey` 函数也允许我们通过按任意键来中断读取的过程。如果有按键发生，那么该函数返回按键的 ASCII 码。如果 `cv::waitKey` 的延迟参数设置的是 0，那么它将无限制地等待用户按键，这在需要通过逐帧检验结果时非常有用。

最后一句话调用 `release` 方法关闭视频文件。然而 `release` 方法将由析构函数调用，因此这次调用不是必需的。

为了打开视频文件，计算机中必须安装有对应的解码器，否则 `cv::VideoCapture` 将无法理解视频格式。通常，如果你能用视频播放器（Windows Media Player）打开该文件，那么 OpenCV 也能读取它。

扩展阅读

你也可以读取与计算机相连的摄像头（例如，USB 摄像头）所捕捉到的视频流。此时你需要指定的是整数的 ID 号，而不是文件名。ID 为 0 将打开默认的摄像头，为了能够停止无止境的摄像头视频流，我们将使用 `cv::waitKey` 函数。

参 考

本章中的“写入视频序列”一节有更多关于视频编解码的信息。

ffmpeg.org 站点提供了完整的开源及跨平台解决方案，可用于音频 / 视频读取、录制、转换以及流媒体。OpenCV 中用于操作视频文件的类正是基于 ffmpeg。

Xvid.org 站点提供了基于 MPEG-4 视频压缩标准的开源编解码库。Xvid 也有名为 DivX 的竞争者，后者提供私有但免费的编解码工具。

10.3 处理视频帧

在本节，我们的目标是对于每个视频帧都应用一些处理函数。我们将在自己的类中封装 OpenCV 的视频获取框架，它将允许我们指定每帧调用的处理函数。

实现方法

我们希望指定一个回调处理函数，每一帧中都将调用它。该函数接受一个 cv::Mat 对象，并且输出处理后 cv::Mat 对象。因此它的函数签名如下：

```
void processFrame(cv::Mat& img, cv::Mat& out);
```

作为这样一个处理函数的例子，下面的 canny 函数计算输入图像的 Canny 边缘：

```
void canny(cv::Mat& img, cv::Mat& out) {
    // 灰度转换
    if(img.channels() == 3)
        cv::cvtColor(img, out, CV_BGR2GRAY);
    // 计算 Canny 边缘
    cv::Canny(out, out, 100, 200);
    // 反转图像
    cv::threshold(out, out, 128, 255, cv::THRESH_BINARY_INV);
}
```

接着，定义一个视频处理类，它将与一个回调函数相关联。使用该类，我们将创建一个实例，指定输入的视频文件，绑定回调函数，然后开始处理过程。依次调用的方法如下：

```
// 创建实例
VideoProcessor processor;
// 打开视频文件
processor.setInput("../bike.avi");
```

```
// 声明显示窗口
processor.displayInput("Current Frame");
processor.displayOutput("Output Frame");
// 以原始帧率播放视频
processor.setDelay(1000./processor.getFrameRate());
// 设置处理回调函数
processor.setFrameProcessor(canny);
// 开始处理过程
processor.run();
```

既然我们已经定义了该类使用的方式，下一步是描述它的实现细节。不难推测出，它将包含多个变量以控制视频帧处理中的不同部分：

```
class VideoProcessor{
private:
    //OpenCV 视频捕捉对象
    cv::VideoCapture capture;
    // 每帧调用的回调函数
    void(*process) (cv::Mat&,cv::Mat&);
    // 确定是否调用回调函数的 bool 变量
    bool callIt;
    // 输入窗口的名称
    std::string windowNameInput;
    // 输出窗口的名称
    std::string windowNameOutput;
    // 延迟
    int delay;
    // 已处理的帧数
    long fnumber;
    // 在该帧数停止
    long frameToStop;
    // 是否停止处理
    bool stop;
public:
    VideoProcessor():callIt(true),delay(0),
        fnumber(0),stop(false),frameToStop(-1){}
```

第一个成员是 `cv::VideoCapture` 对象，第二个是处理函数的指针。这将通过对应的设置函数进行赋值：

```
// 设置回调函数
void setFrameProcessor(
    void(*frameProcessingCallback) (cv::Mat&, cv::Mat&)) {
    process= frameProcessingCallback;
}
```

下面的方法将用于打开视频文件：

```
// 设置视频文件的名称
bool setInput(std::string filename) {
    fnumber= 0;
    // 释放之前打开过的资源
    capture.release();
    images.clear();
    // 打开视频文件
    return capture.open(filename);
}
```

当处理视频帧时显示它们的内容通常很有用，因此我们增加了两个函数用于创建显示窗口：

```
// 创建输入窗口
void displayInput(std::string wn) {
    windowNameInput= wn;
    cv::namedWindow(windowNameInput);
}

// 创建输出窗口
void displayOutput(std::string wn) {
    windowNameOutput= wn;
    cv::namedWindow(windowNameOutput);
}

// 不再显示处理后的帧
void dontDisplay() {
    cv::destroyWindow(windowNameInput);
    cv::destroyWindow(windowNameOutput);
    windowNameInput.clear();
    windowNameOutput.clear();
}
```

如果没有调用这两个函数，那么对应的帧不会被显示。名为 run 的主要函数包含

提取视频帧的循环：

```
// 获取并处理序列帧
void run() {
    // 当前帧
    cv::Mat frame;
    // 输出帧
    cv::Mat output;
    // 打开失败时
    if(!isOpened())
        return;
    stop= false;
    while(!isStopped()) {
        // 读取下一帧
        if(!readNextFrame(frame))
            break;
        // 显示输出帧
        if(windowNameInput.length()!=0)
            cv::imshow(windowNameInput,frame);
        // 调用处理函数
        if(callIt) {
            // 处理当前帧
            process(frame,output);
            // 增加帧数
            fnumber++;
        }else{
            output= frame;
        }
        // 显示输出帧
        if(windowNameOutput.length()!=0)
            cv::imshow(windowNameOutput,output);
        // 引入延迟
        if(delay>=0&& cv::waitKey(delay)>=0)
            stopIt();
        // 检查是否需要停止运行
        if(frameToStop>=0 &&getFrameNumber()==frameToStop)
            stopIt();
    }
}
```

```

// 停止运行
void stopIt(){
    stop= true;
}
// 是否已停止？
bool isStopped(){
    return stop;
}
// 是否开始了捕获设备？
bool isOpened(){
    capture.isOpened();
}
// 设置帧间的延迟
//0 意味着在每帧都等待用户按键
//负数意味着没有延迟
void setDelay(int d){
    delay= d;
}

```

下面的方法是用私有方法以读取视频帧：

```

// 得到下一帧
// 可能是：视频文件或摄像头
bool readNextFrame(cv::Mat& frame){
    return capture.read(frame);
}

```

有时只需要简单地打开并播放视频文件，不需要调用回调函数。因此我们提供两个方法以指定是否需要每帧调用回调函数：

```

// 需要调用回调函数
void callProcess(){
    callIt= true;
}
// 不需要调用回调函数
void dontCallProcess(){
    callIt= false;
}

```

最后，我们可以在特定的帧停止：

```
void stopAtFrameNo(long frame) {  
    frameToStop= frame;  
}  
  
// 返回下一帧的帧数  
long getFrameNumber() {  
    // 得到捕获设备的信息  
    long fnumber= static_cast<long>(  
        capture.get(CV_CAP_PROP_POS_FRAMES));  
    return fnumber;  
}
```

如果该类用于运行开始时展示的代码片段，那么两个窗口将分别播放输入视频以及处理后的结果，它们将保留原始视频中的帧率（通过 `setDelay` 方法指定了延迟）。图 10.2 是输入视频的一帧，对应的输出帧如图 10.3 所示。

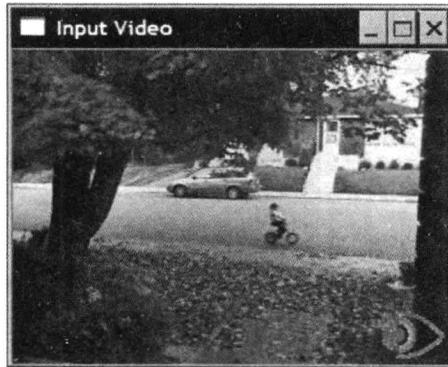


图 10.2

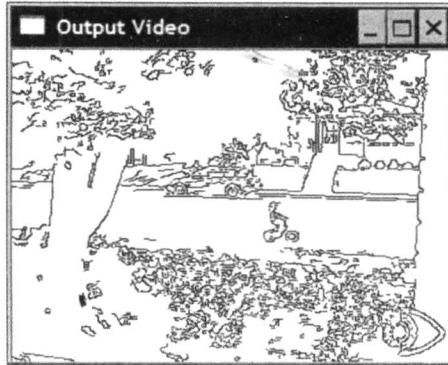


图 10.3

作用原理

正如我们在其他节中所做的，我们的目标是创建一个类来封装视频处理算法的常用功能。在这个类中，视频捕获循环由 `run` 方法实现，它通过调用 `cv::VideoCapture` 类的 `read` 方法进行帧提取，之后进行一系列的操作。但是在这之前，需要检查是否成功获得图像帧。输入窗口仅在使用 `displayInput` 函数指定窗口名称之后才会显示。回调函数仅在使用 `setFrameProcessor` 进行指定后才会被调用。输出窗口仅在使用 `displayOutput` 函数指定窗口名称之后才会显示。延迟仅在使用 `setDelay` 进行指定后才会发生。最后，如果通过 `stopAtFrameNo` 方法定义了停止帧后，我们还将检查当前帧的数目。

该类同时包含一些取值函数与设置函数，它们仅仅是对 `cv::VideoCapture` 框架的简单封装。

扩展阅读

`VideoProcessor` 类可以帮助部署视频处理模块，可以进行一些额外的改进。

1. 处理图像序列

有时输入序列是由存储在不同文件中的一系列图像组成的，我们可以修改类来适应这类输入。你需要添加一个成员变量以保存图像文件名称的数组，以及对应的迭代器：

```
// 把图像文件名的数组作为输入
std::vector<std::string>images;
// 图像向量的迭代器
std::vector<std::string>::const_iterator itImg;
```

`setInput` 函数用于指定读取的文件名：

```
// 设置输入的图像向量
bool setInput(const std::vector<std::string>&imgs) {
    fnumber= 0;
    // 释放之前打开过的资源
    capture.release();
    // 输入将是该图像的向量
    images= imgs;
    itImg= images.begin();
    return true;
}
```

isOpened 方法变为

```
// 视频捕捉设备是否已开启
bool isOpened(){
    return capture.isOpened() || !images.empty();
}
```

最后一个需要修改的是私有的 readNextFrame 方法，根据 images 数组的大小它将选择是否从文件名数组中读取图像。 setInput 方法如果指定的是视频文件，那么该数组将被清零：

```
// 得到下一帧
// 可以是：视频文件；摄像头；图像数组
bool readNextFrame(cv::Mat& frame){
    if(images.size()==0)
        return capture.read(frame);
    else{
        if(itImg != images.end()){
            frame= cv::imread(*itImg);
            itImg++;
            return frame.data != 0;
        }else{
            return false;
        }
    }
}
```

2. 使用帧处理类

在面向对象的上下文中，更适合使用帧处理类而不是一个帧处理函数。事实上，使用类给予程序员在设计算法方面更多的灵活度。VideoProcessor 内部使用的帧处理类如下：

```
// 帧处理器的接口
class FrameProcessor{
public:
    // 处理方法
    virtual void process(cv::Mat &input, cv::Mat &output)=0;
};
```

VideoProcessor 中的 setFrameProcessor 方法用于设置 FrameProcessor 指针：

```
// 设置 FrameProcessor 实例
void setFrameProcessor(FrameProcessor*frameProcessorPtr)
{
    // 使回调函数无效化
    process= 0;
    // 重新设置 FrameProcessor 实例
    frameProcessor= frameProcessorPtr;
    callProcess();
}
```

当指定 FrameProcessor 实例时，将使得之前可能设置过的回调函数无效化。同样的，在设置回调函数时，也会使之前可能设置过的 FrameProcessor 实例无效化：

```
// 设置回调函数
void setFrameProcessor(
    void(*frameProcessingCallback) (cv::Mat&,cv::Mat&)) {
    // 使 FrameProcessor 实例无效化
    frameProcessor= 0;
    // 重新设置回调函数
    process= frameProcessingCallback;
    callProcess();
}
```

run 方法的 while 循环也进行修改：

```
while(!isStopped()){
    // 读取下一帧
    if(!readNextFrame(frame))
        break;
    // 显示输出帧
    if(windowNameInput.length()!=0)
        cv::imshow(windowNameInput,frame);
    /** 调用帧处理实例或回调函数 */
    if(callIt){
        // 处理当前帧
        if(process)// 如果是回调函数
            process(frame,output);
```

```
else if(frameProcessor)
    // 如果是帧处理实例
    frameProcessor->process(frame,output);
    // 增加帧数
    fnumber++;
} else{
    output= frame;
}
// 显示输出帧
if(windowNameOutput.length()!=0)
    cv::imshow(windowNameOutput,output);
// 引入延迟
if(delay>=0&& cv::waitKey(delay)>=0)
    stopIt();
// 检查是否需要停止运行
if(frameToStop>=0&& getFrameNumber()==frameToStop)
    stopIt();
}
```

参 考

本章的“跟踪视频中的特征点”秘诀将展示如何使用 FrameProcessor 接口。

10.4 写入视频序列

在之前的节中，我们学习到如何读取视频文件，并提取它的图像帧。本则秘诀将介绍如何创建视频文件并写入图像帧。这将允许我们完成典型的视频处理链：读取输入视频流、处理它的图像帧，最后存储结果到视频文件中。

实现方法

我们将扩展 VideoProcessor 类以给予它写视频文件的能力。这通过 OpenCV 的 cv::VideoWriter 类得到实现，我们增加这么一个成员变量（以及一些额外的变量）：

```
class VideoProcessor{
private:
    ...
}
```

```
//OpenCV 的写视频对象  
cv::VideoWriter writer;  
// 输出文件名称  
std::string outputFile;  
// 输出图像的当前索引  
int currentIndex;  
// 输出图像名称中的数字位数  
int digits;  
// 输出图像的扩展名  
std::string extension;
```

额外的方法被用于指定且打开输出视频文件：

```
// 设置输出视频文件  
// 默认使用于输入视频相同的参数  
bool setOutput(const std::string &filename,  
                int codec=0,double framerate=0.0,  
                bool isColor=true){  
    outputFile= filename;  
    extension.clear();  
    if(framerate==0.0)  
        framerate=getFrameRate();// 与输入相同  
    char c[4];  
    // 使用相同的编码格式  
    if(codec==0){  
        codec=getCodec(c);  
    }  
    // 打开输出视频  
    return writer.open(outputFile,// 文件名  
                      codec,           // 将使用的编码格式  
                      framerate,       // 帧率  
                      getFrameSize(), // 尺寸  
                      isColor);        // 是否是彩色视频 ?  
}
```

一旦我们打开了视频，便重复调用 `cv::VideoWriter` 类的 `write` 方法将图像帧写入文件。正如我们在之前节中展示的，我们也希望用户能够写图像帧到单独的图像文件中。因此，私有方法 `writeNextFrame` 处理这两种可能的情况：

```
// 输出帧
// 可能是：视频文件或图像文件
void writeNextFrame(cv::Mat& frame) {
    if(extension.length()){// 我们输出到图像文件
        std::stringstream ss;
        // 组成输出文件名称
        ss<<outputFile<<std::setfill('0')
           <<std::setw(digits)
           <<currentIndex++<<extension;
        cv::imwrite(ss.str(),frame);
    }else{// 我们输出到视频文件
        writer.write(frame);
    }
}
```

对于输出是由独立的图像文件组成的情况，我们需要额外的设置方法：

```
// 设置输出未独立的图像文件
// 扩展名必须是 ".jpg", ".bmp"...
bool setOutput(const std::string &filename,// 前缀
               const std::string &ext,           // 后缀
               int numberofDigits=3,           // 数字位数
               int startIndex=0){             // 开始索引
    // 数字位数必须是正的
    if(numberOfDigits<0)
        return false;
    // 文件名及其后缀
    outputFile= filename;
    extension= ext;
    // 文件名命名中的数字位数
    digits= numberofDigits;
    // 开始索引
    currentIndex= startIndex;
    return true;
}
```

run 方法的视频捕获循环中增加一个新的步骤：

```
while(!isStopped()) {
    // 读取下一帧
    if(!readNextFrame(frame))
        break;
    // 显示输出帧
    if(windowNameInput.length()!=0)
        cv::imshow(windowNameInput, frame);
    // 调用处理函数或方法
    if(callIt) {
        // 处理当前帧
        if(process)
            process(frame, output);
        else if(frameProcessor)
            frameProcessor->process(frame, output);
        // 增加帧数
        fnumber++;
    } else{
        output= frame;
    }
    //** 输出图像序列 **
    if(outputFile.length()!=0)
        writeNextFrame(output);
    // 显示输出帧
    if(windowNameOutput.length()!=0)
        cv::imshow(windowNameOutput, output);
    // 引入延迟
    if(delay>=0 && cv::waitKey(delay)>=0)
        stopIt();
    // 检查是否需要停止运行
    if(frameToStop>=0 &&
        getFrameNumber ()==frameToStop)
        stopIt();
}
}
```

下例中是一个简单的程序，实现了视频的读取与处理，并将结果写到视频文件中：

```
// 创建实例  
VideoProcessor processor;  
// 打开视频文件  
processor.setInput("../bike.avi");  
processor.setFrameProcessor(canny);  
processor.setOutput("../bikeOut.avi");  
// 开始处理过程  
processor.run();
```

如果你希望结果被保存到一系列图像中，那么可以将先前的语句修改为

```
processor.setOutput("../bikeOut",".jpg");
```

使用默认的数位数（3）及开始索引（0），将创建文件 bikeOut000.jpg、bikeOut001.jpg、bikeOut002.jpg 等等。

作用原理

当视频被写到文件中时，它是通过一种视频编解码进行保存的。编解码是一个软件模块，能够编码和解码视频流，它同时定义了文件格式及保存信息的压缩方式。很明显的，视频解码时必须使用相同的编码格式。因此，我们使用四个字符来标示编解码。当软件需要写入视频文件时，通过读取指定的四个字符可以确定将使用的编码格式。

正如名字所示，这四个 ASCII 字符通过相加能够转换为一个整数。对于已经打开的 cv::VideoCapture 对象使用 get 方法的 CV_CAP_PROP_FOURCC 标志位，可以获取视频文件的编解码。我们在 VideoProcessor 类中定义一个方法以获取输入视频的编解码：

```
// 得到输入视频的编解码  
int getCodec(char codec[4]) {  
    // 未指定图像数组  
    if(images.size() != 0) return -1;  
    union{ // 4-char 编码的数据结果  
        int value;  
        char code[4]; } returned;  
    // 得到编码  
    returned.value = static_cast<int>(capture.get(CV_CAP_PROP_FOURCC));  
    // 得到 4 个字符
```

```

    codec[0]=returned.code[0];
    codec[1]=returned.code[1];
    codec[2]=returned.code[2];
    codec[3]=returned.code[3];
    // 返回对应的整数
    return returned.value;
}

```

get 方法总是返回 double 数值，需要转换成整数，之后通过 union 数据结果可以提取出四个字符。如果我们打开的是测试视频序列，那么通过下列语句：

```

char codec[4];
processor.getCodec(codec);
std::cout << "Codec:" << codec[0] << codec[1]
             << codec[2] << codec[3] << std::endl;

```

我们将得到：

Codec:XVID

当写入到视频文件时，编解码必须通过四字符形式指定。这是 cv::VideoWriter 类中 open 方法的第二个参数。例如，你可以使用与输入视频相同的编解码，这正是 setOutput 方法中的默认选项。你也可以传递 -1，这将弹出窗口要求你从可选的编解码中挑选一个，如图 10.4 所示。

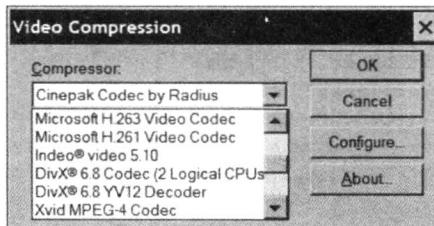


图 10.4

你在窗口中看到的列表对应于计算机中安装的编解码。被选中的编解码将自动应用于 open 方法。

10.5 跟踪视频中的特征点

本节是关于读、写、处理视频序列的内容，目标是能够分析一个完整的视频序列。

作为一个范例，你将在本节中学习到如何通过对视频进行时序分析，在帧与帧之间跟踪特征点。

实现方法

在开始跟踪前，首先要在初始帧中检测特征点，之后在下一帧中尝试跟踪这些点。你必须找到新的图像帧中这些点的位置。很明显的，由于我们处理的是视频序列，很有可能特征点所在的物体已经移动过（运动也有可能是相机引起的）。因此，你必须在特征点的先前位置附近进行搜索，以找到下一帧中它的新位置。这正是 cv::calcOpticalFlowPyrLK 函数所实现的工作。你输入两个连续的图像帧以及第一幅图像中检测到的特征点数组，该函数将返回一组新的特征点为位置。为了跟踪完整的序列，你需要在帧与帧之间重复这个过程，不可避免地你也会丢失其中一些点，于是被跟踪的特征点数目会减少。为了解决这个问题，我们可以不时地检测新的特征值。

现在之前实现的框架的优点便发挥出来了，我们将定义一个类来实现本章中介绍的 FrameProcessor 接口，类中新增加的变量用于检测及跟踪特征点：

```
class FeatureTracker:public FrameProcessor{
    cv::Mat gray;           // 当前灰度图像
    cv::Mat gray_prev;       // 之前灰度图像
    // 两幅图像间跟踪的特征点 0->1
    std::vector<cv::Point2f>points[2];
    // 跟踪的点初始位置
    std::vector<cv::Point2f>initial;
    std::vector<cv::Point2f>features;           // 检测到的特征
    int max_count;          // 需要跟踪的最大特征数目
    double qlevel;           // 特征检测中的质量等级
    double minDist;          // 两点之间的最小距离
    std::vector<uchar>status; // 检测到的特征的状态
    std::vector<float>err;      // 跟踪过程中的错误
public:
    FeatureTracker():max_count(500),qlevel(0.01),minDist(10.){}
```

接着，我们定义 process 方法，它将在每一帧被调用，基本调用次序如下。首先，在需要时检测特征点。接着，跟踪这些点，将无法跟踪或是不再希望跟踪的点去除掉。现在你准备好处理被成功跟踪的点了。最后，当前帧以及它的点在下一次迭代中成为

之前的帧以及之前的点。以下是具体实现方法：

```
void process(cv::Mat &frame, cv::Mat &output) {
    // 转换为灰度图像
    cv::cvtColor(frame, gray, CV_BGR2GRAY);
    frame.copyTo(output);
    //1. 如果需要添加新的特征点
    if(addNewPoints())
    {
        // 进行检测
        detectFeaturePoints();
        // 添加检测到的特征到当前跟踪的特征中
        points[0].insert(points[0].end(), features.begin(), features.end());
        initial.insert(initial.end(), features.begin(), features.end());
    }
    // 对于序列中的第一幅图像
    if(gray_prev.empty())
        gray.copyTo(gray_prev);
    //2. 跟踪特征点
    cv::calcOpticalFlowPyrLK(
        gray_prev, gray, // 两幅连续图
        points[0], // 图 1 中的输入点坐标
        points[1], // 图 2 中的输出点坐标
        status, // 跟踪成功
        err); // 跟踪错误
    //2. 遍历所有跟踪的点进行筛选
    int k=0;
    for(int i=0;i<points[1].size();i++){
        // 是否需要保留该点？
        if(acceptTrackedPoint(i)){
            // 进行保留
            initial[k]= initial[i];
            points[1][k++]= points[1][i];
        }
    }
    // 去除不成功的点
    points[1].resize(k);
    initial.resize(k);
    //3. 处理接受的跟踪点
```

```
    handleTrackedPoints(frame, output);
    //4. 当前帧的点和图像变为前一帧的点和图像
    std::swap(points[1], points[0]);
    cv::swap(gray_prev, gray);
}
```

该方法使用了四个其他的辅助函数。你可以轻易改变任何一个方法以定义你自己的跟踪器的行为。这些方法中的第一个进行的是检测特征点，我们在第 8 章的 8.1 节中已经讨论过 `cv::goodFeatureToTrack` 函数：

```
// 检测特征点
void detectFeaturePoints() {
    // 检测特征
    cv::goodFeaturesToTrack(gray, // 图像
                           features,      // 检测到的特征
                           max_count,     // 特征的最大数目
                           qlevel,        // 质量等级
                           minDist);      // 两个特征之间的最小距离
}
```

第二个方法决定是否需要检测新的特征点：

```
// 是否需要添加新的点
bool addNewPoints() {
    // 如果点的数量太少
    return points[0].size()<=10;
}
```

第三个方法根据应用定义的标准移除一些跟踪点，这里我们决定移除不再移动的点，加上 `cv::calcOpticalFlowPyrLK` 函数无法跟踪的点：

```
// 决定哪些点应该跟踪
bool acceptTrackedPoint(int i) {
    return status[i]&&
           // 如果它移动了
           (abs(points[0][i].x-points[1][i].x) +
            (abs(points[0][i].y-points[1][i].y))>2);
}
```

最后，第四个方法将所有跟踪点绘制在当前帧上，并用直线将它们与初始位置相

连（即它们在最初检测到的位置）：

```
// 处理当前跟踪的点
void handleTrackedPoints(cv::Mat &frame,
                           cv::Mat &output) {
    // 遍历所有跟踪点
    for(int i = 0;i<points[1].size();i++){
        // 绘制直线与圆
        cv::line(output,
                  initial[i],      // 初始位置
                  points[1][i],    // 新位置
                  cv::Scalar(255,255,255));
        cv::circle(output,points[1][i],3,cv::Scalar(255,255,255),-1);
    }
}
```

下面这个简单的 main 函数用于跟踪视频序列中的特征点：

```
int main()
{
    // 创建视频处理器实例
    VideoProcessor processor;
    // 创建特征跟踪器实例
    FeatureTracker tracker;
    // 打开视频文件
    processor.setInput("../bike.avi");
    // 设置帧处理器对象
    processor.setFrameProcessor(&tracker);
    // 声明显示窗口
    processor.displayOutput("Tracked Features");
    // 以原始帧率播放视频
    processor.setDelay(1000./processor.getFrameRate());
    // 开始处理过程
    processor.run();
}
```

生成的程序将展示出随时间变化的特征点，例如，此处便是两个不同时刻的图像帧。在这个视频中，相机的位置是规定的，因此年轻的骑手是唯一的运动物体。图 10.5 是视频开始时候的图像帧。

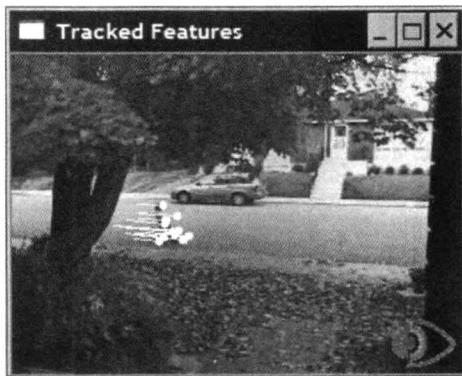


图 10.5

几秒后，我们得到图 10.6。

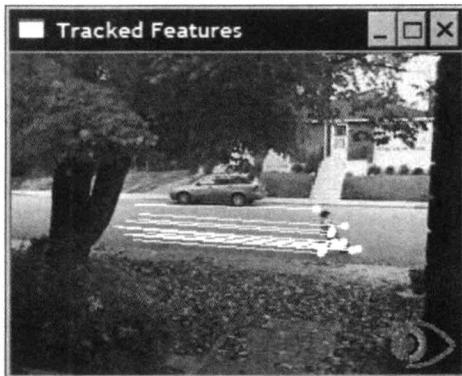


图 10.6

作用原理

为了在不同帧之间跟踪特征点，我们必须定位特征点在随后的图像帧中的新位置。如果我们假设特征点的强度在相邻帧之间不发生变化，那么我们要找到一个偏移量 (u , v) :

$$I_t(x, y) = I_{t+1}(x+u, y+v)$$

其中， I_t 和 I_{t+1} 分别是当前帧以及下一时刻的值。这个强度不变的假设对于拍摄时间相近的图像中的微小偏移量而言都是成立的。我们可以使用泰勒展开式来近似这个方程式，新的方程式将涉及图像的倒数：

$$I_{t+1}(x+u, y+v) \approx I_t(x, y) + \frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v + \frac{\partial I}{\partial t}$$

如果强度不变假设成立，可以推出另一个方程式：

$$\frac{\partial I}{\partial x} u + \frac{\partial I}{\partial y} v = - \frac{\partial I}{\partial t}$$

这个著名的约束便是基础光流约束方程式，在引入另一个假设后，它被使用于 Lukas-Kanade 特征跟踪算法，即特征点附近所有点的偏移量都是一致的。因此我们对于拥有独特未知偏移量 (u, v) 的点都利用光流约束。这给予我们超过未知数的数量（两个）的方程式，我们可以使用均方法进行求解。在实践中它通过迭代的方式进行求解，OpenCV 实现了在不同分辨率进行估算求解的函数，使得可以更有效地进行搜索，同时也能适应较大的偏差。默认情况下，图像的层数是 3，而搜索窗口的尺寸是 15。这些参数是可以修改的，你可以指定终止条件来控制迭代过程。`cv::calcOpticalFlowPyrLK` 函数的第 6 个参数包含了剩余的均方差，可用来评估跟踪的质量。第 5 个参数包含二值的标志位，告诉我们对应点的跟踪是否成功。

上面描述了 Lukas-Kanade 跟踪器的基本原理。当前的实现也包含其他的优化及改进，使得算法在处理大量特征点的位移上更加高效。

参 考

第 8 章介绍了特征点的检测。

B.Lucas 及 T.Kanade 的经典论文 “*An iterative Image Registration Technique with an Application to Stereo Vision*” (*Int. Joint Conference in Artificial Intelligence*, pp.674-679, 1981) 描述了最初的特征点跟踪算法。

J.Shi 及 C.Tomasi 的论文 “*Good Features to Track*” (*IEEE Conference on Computer Vision and Pattern Recognition*, pp.593-600, 1994) 对原跟踪算法进行了改进。

10.6 提取视频中的前景物体

当观察场景的是一个固定的相机，背景几乎保持不变。在这种情况下，感兴趣的元素是在场景中运动的物体。为了提取出这些前景物体，我们需要对背景建模，然后将当前帧的模型与背景模型进行比较，以检测前景物体。这正是我们将在本则秘诀中实现的，前景提取是智能监控应用的基础步骤。

实现方法

如果我们能够获取当前场景背景的图像（即不包含前景物体），那么通过简单的图像差分便可以提取出前景：

```
// 计算当前图像与背景的差异
cv::absdiff(backgroundColor, currentImage, foreground);
```

差异足够大的像素都将被视为前景像素。然而，大部分情况下背景图像不是立即可用的。事实上，很难确保特定图像中不包含任何前景物体。在繁忙的场景中，几乎不会发生这种情况。背景通常会随着时间变化，例如，从日出到日落之间光照条件会发生变化，同时新的物体可能会进入或离开背景。

因此，我们必须能够动态创建背景的模型，这可以通过长时间观察场景来实现。如果我们假设通常情况背景在每个像素位置都是可见的，那么对所有观察值进行平均便是一个不错的方法。但是这有很多局限性。首先，这需要在计算背景前保存大量的图像。其次，在累加图像时无法提取前景物体。这个方法也带来新的问题，为了计算可以满意的背景模型，需要累加多少图像，并且从何时开始。另外，如果图像中给定的像素总是在观察一个前景物体，也将对平均后的背景造成影响。

一个更好的策略是定期更新背景模型，这可以通过计算滑动平均值，即对时序信号计算均值时考虑接收到的最新值。如果 p_t 是 t 时刻的像素值， μ_{t-1} 是当前的平均值，那么新的平均值将是

$$\mu_t = (1-\alpha) \mu_{t-1} + \alpha p_t$$

参数 α 被称为学习率，它定义了当前值对平均值的影响程度。该值越大，滑动平均值对观察值的适应速度更快。为了构建背景模型，需要对图像帧的每个像素计算滑动平均数。判断前景要素的依据是当前图像与背景模型之间的差值。

我们将创建一个封装类：

```
class BGFGSegmentor:public FrameProcessor{
    cv::Mat gray;           // 当前灰度图像
    cv::Mat background;     // 累积的背景
    cv::Mat backImage;       // 背景图像
    cv::Mat foreground;      // 前景图像
    // 背景累加中的学习率
    double learningRate;
```

```
int threshold; // 前景提取的阈值
public:
    BGFGSegmentor():threshold(10),learningRate(0.01){}
```

主要的运算过程包含比较当前帧与背景模型，并且更新模型：

```
// 处理方法
void process(cv::Mat& frame, cv::Mat &output) {
    // 转换为灰度图像
    cv::cvtColor(frame,gray,CV_BGR2GRAY);
    // 初始化背景为第一帧
    if(background.empty())
        gray.convertTo(background,CV_32F);
    // 转换背景图像为 8U 格式
    background.convertTo(backImage,CV_8U);
    // 计算差值
    cv::absdiff(backImage,gray,foreground);
    // 应用阈值化到前景图像
    cv::threshold(foreground,output,
                  threshold,255,cv::THRESH_BINARY_INV);
    // 对背景累加
    cv::accumulateWeighted(gray,background,learningRate,output);
}
```

基于我们的视频处理框架，前景提取算法可以如下构建：

```
int main()
{
    // 创建视频处理器实例
    VideoProcessor processor;
    // 创建背景 / 前景分段器
    BGFGSegmentor segmentor;
    segmentor.setThreshold(25);
    // 打开视频文件
    processor.setInput("../bike.avi");
    // 设置帧处理器
    processor setFrameProcessor(&segmentor);
    // 声明显示窗口
    processor.displayOutput("Extracted Foreground");
    // 以原始帧率播放视频
    processor.setDelay(1000./processor.getFrameRate());
```

```
// 开始处理过程  
processor.run();  
}
```

生成的二值前景图像将如图 10.7 所示。

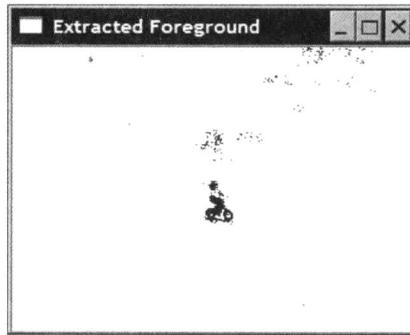


图 10.7

作用原理

`cv::accumulateWeighted` 函数实现了滑动平均值计算的功能，它生成的图像是浮点数格式，因此在进行比较前需要进行格式转换。依次调用 `cv::absdiff` 与 `cv::threshold` 能够生成阈值化的差异绝对值，并提取出前景图像。我们使用该前景图像作为 `cv::accumulateWeighted` 的掩码以避免更新被认为是前景的像素。我们的图像中像素在前景处为 `false`（即 0），因此在生成的图像中前景物体显示为黑色。

最后，出于简洁，背景模型是基于灰度图像。维护彩色版的图像需要更多的计算量，并且也不会显著提升算法质量。算法的难点在于找到合适的阈值。

扩展阅读

先前的方案对于背景相对稳定的简单场景比较有效，然而在许多情况下，背景也许会在部分区域震荡变化，频繁地检测到错误的前景。这也许是由于运动的背景物体（如树叶），或是耀眼的效果（如水面）。为了处理这些问题，人们提出了更复杂的背景建模方法。

混合高斯算法

混合高斯（Mixture of Gaussian）算法便是其中之一，它的处理过程与之前提到的

很类似，同时有以下变化。

首先，该方法对每个像素维护多个模型（即多个滑动平均值）。因此，如果背景像素在两个值之间来回变化，那么将存储两个滑动平均值。如果新的像素值不属于其中之一，我们才会认为它是前景值。

其次，不仅仅保存滑动平均值，还保存滑动方差。它的计算方式如下：

$$\sigma_t^2 = (1-\sigma)\sigma_{t-1}^2 + \sigma(p_t - \mu_t)^2$$

计算得到的均值与方差形成了一个高斯模型，通过它我们可以获知某个像素值属于该模型的概率。它使得确定合适的阈值变得简单，因为现在它表现为一个概率而非差异的绝对值。同时，在背景中变化较明显的区域，前景物体需要的差值将大得多。

最后，当高斯模型的击中频率不够频繁，它会从当前的模型中移除。反之，如果一个像素值位于当前维护的背景模型之外（即它是一个前景像素），那么新的高斯模型将被创建。如果之后它被频繁击中，那么会成为背景的一部分。

这个复杂算法比我们的简单背景/前景分割器实现难度更高。幸运的是，OpenCV 中的 `cv::BackgroundSubtractorMOG` 类实现了该功能，并且定义为更通用的 `cv::BackgroundSubtractor` 类的子类。当使用默认参数时，它的用法非常简单：

```
int main()
{
    // 打开视频文件
    cv::VideoCapture capture("../bike.avi");
    // 检查视频是否成功打开
    if(!capture.isOpened())
        return 0;
    // 当前视频帧
    cv::Mat frame;
    // 前景图像
    cv::Mat foreground;
    cv::namedWindow("Extracted Foreground");
    // 使用默认参数的 Mixture of Gaussian 对象
    cv::BackgroundSubtractorMOG mog;
    bool stop(false);
    // 遍历每一帧
    while(!stop) {
        // 读取下一帧
        if(!capture.read(frame))
```

```
        break;
    // 更新背景并返回前景
    mog(frame,foreground,0.01);
    // 对图像取反
    cv::threshold(foreground,foreground,
                  128,255,cv::THRESH_BINARY_INV);
    // 显示前景
    cv::imshow("Extracted Foreground",foreground);
    // 引入延迟
    // 也可通过按键停止
    if(cv::waitKey(10)>=0)
        stop=true;
    }
}
```

正如我们所看到的，需要创建一个实例然后调用方法，将同时更新背景模型以及返回前景图像（额外的参数是学习率）。显示的分段图像如图 10.8 所示。

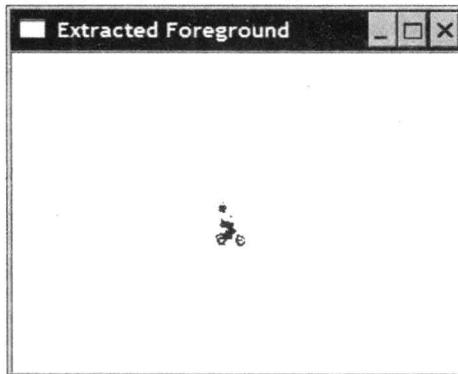


图 10.8

该类的一个参数是每个像素可能拥有的高斯模型数量。

参 考

C.Stauffer 及 W.E.L.Grimson 的论文 “*Adaptive Background Mixture Models for Real-time Tracking*” (*Conf.on Computer Vision and Pattern Recognition*, 1999) 详细描述了混合高斯算法。

[General Information]

书名=OpenCV 2 计算机视觉编程手册

作者=(加)ROBERT LAGANIÈRE著；张静译

页数=244

SS号=13340922

出版日期=2013.07

出版社=科学出版社

ISBN号=978-7-03-037581-0

中图法分类号=TP391.41-62

尺寸=26cm

原书定价=45.00

参考文献格式=Robert Laganière著. OpenCV2计算机视觉编程手册. 北京市：科学出版社, 2013.06.

内容提要=OpenCV提供的视觉处理算法非常丰富，并且以C语言编写，加上其开源的特性，处理得当，不需要添加新的外部支持也可以完整的编译链接生成执行程序，所以很多人用它来做算法的移植。OpenCV的代码经过适当改写可以正常的运行在DSP系统和单片机系统中，目前这种移植在大学中经常作为相关专业本科生毕业设计或者研究生的课题。

封面

书名

版权

前言

目录

第1章 接触图像

1.1引言

1.2安装OpenCV库

1.3使用MS Visual C++创建OpenCV工程

1.4使用Qt创建OpenCV项目

1.5载入、显示及保存图像

1.6使用Qt创建GUI应用

第2章 操作像素

2.1引言

2.2存取像素值

2.3使用指针遍历图像

2.4使用迭代器遍历图像

2.5编写高效的图像遍历循环

2.6遍历图像和邻域操作

2.7进行简单的图像算术

2.8定义感兴趣区域

第3章 基于类的图像处理

3.1引言

3.2在算法设计中使用策略(Strategy)模式

3.3使用控制器(Controller)实现模块间通信

3.4使用单件(Singleton)设计模式

3.5使用模型-视图-控制器(Model-View-Controller)架构设计应用程序

3.6颜色空间转换

第4章 使用直方图统计像素

4.1引言

4.2计算图像的直方图

4.3使用查找表修改图像外观

4.4直方图均衡化

4.5反投影直方图以检测特定的图像内容

4.6使用均值漂移(Mean Shift)算法查找物体

4.7通过比较直方图检索相似图片

第5章 基于形态学运算的图像变换

5.1引言

5.2使用形态学滤波对图像进行腐蚀、膨胀运算

5.3使用形态学滤波对图像进行开闭运算

5.4使用形态学滤波对图像进行边缘及角点检测

5.5使用分水岭算法对图像进行分割

5.6使用GrabCut算法提取前景物体

第6章 图像滤波

6.1引言

6.2使用低通滤波器

6.3使用中值滤波器

6.4使用方向滤波器检测边缘

6.5计算图像的拉普拉斯变换

第7章 提取直线、轮廓及连通区域

7.1引言

7.2使用Canny算子检测轮廓

7.3 使用霍夫变换检测直线

7.4用直线拟合一组点

7.5提取连通区域的轮廓

7.6计算连通区域的形状描述符

第8章 检测并匹配兴趣点

8.1引言

8.2检测Harris角点

8.3检测FAST特征

8.4检测尺度不变的SURF特征

8.5描述SURF特征

第9章 估算图像间的投影关系

9.1引言

9.2相机标定

9.3计算一对图像的基础矩阵

9.4使用随机采样一致算法 (RANSAC) 进行图像匹配

9.5计算两幅图之间的单应矩阵

第10章 处理视频序列

10.1引言

10.2读取视频序列

10.3处理视频帧

- 10.4写入视频序列
- 10.5跟踪视频中的特征点
- 10.6提取视频中的前景物体